(2)

USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

Ada Transition Research Project (Phase I)

(ASQB-GI-91-005)

DECEMBER 1990

64pg

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704--188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: NONE
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT: N/A
2b. DECLASSIFICATION / DOUWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): ASQB-GI-91-005	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: AIRMICS; 6b. OFFICE SYMBOL (if applicable): ASQB-GI	7a. NAME OF MONITORING ORGANIZATION: N/A
6c. ADDRESS (City, State, and ZIP Code): 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800	7b. ADDRESS (City, State, and Zip Code): N/A
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AIRMICS; 8b. OFFICE SYMBOL (if applicable): ASQB-GI	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 62783A; PROJECT NO. DY10; TASK NO. 02-04; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Ada Transition Research Project (Phase I) (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Reginald L. Hobbs; Joseph J. Nealon; Richard Wassmuth

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
	FROM ______ TO ______	1990 December 10	85

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computer-Aided Software Engineering (CASE); Distributed Software;Ada; COBOL;Systems Analysis, Systems Design, Life Cycle Development;Functional Decomposition; Object-Oriented Design

19. ABSTRACT (Continue on reverse if necessary and identity by block number)

The objective of this research project was the development of a design strategy using software engineering methodologies to transition STAMIS applications from COBOL to Ada. This is the final report outlining the reverse engineering and redesign strategies for STAMISs as well as significant findings encountered during the actual transition process. This report constitutes the completion of Phase I of the transition project; the coding and implementation of the redesigned STAMIS will occur in succeeding phases.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: [✓ UNCLASSIFIED / UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Reginald L. Hobbs	22b. TELEPHONE (Include Area Code): (404) 894-3110; 22c OFFICE SYMBOL: ASQB-GI

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

This research was performed as an in-house project at the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS) in conjunction with personnel from the Software Development Center-Atlanta (SDC-A). The objective of this initial phase of the project was to evaluate the use of various tools and methods in changing existing applications from batch-oriented COBOL systems to interactive systems written in Ada. Specifically, Phase I of the project examined the application of reverse engineering tools and methods, compared functional decomposition versus object-oriented design, evaluated the use of CASE tools for systems analysis and design, assessed Ada training provided by the Army, and examined the maintainability of code designed using currently available tools. The actual coding, implementation, and maintenance of the redesigned system will occur in succeeding phases. This research report is not to be construed as an official Army position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

s/ [illegible]
Glenn E. Racine
Chief
CISD

s/ [illegible]
John R. Mitchell
Director
AIRMICS

USAISEC

ADA TRANSITION RESEARCH PROJECT (Phase I)

FINAL REPORT

DECEMBER 1990

Prepared By

Reginald L. Hobbs
Joseph J. Nealon
Richard Wassmuth

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

DTIC QUALITY INSPECTED 3

AIRMICS
115 OKEEFE BUILDING
ATLANTA, GA 30332

TABLE OF CONTENTS

Ada Transition Research Project (Phase I)
Final Report

1.0 ABSTRACT:

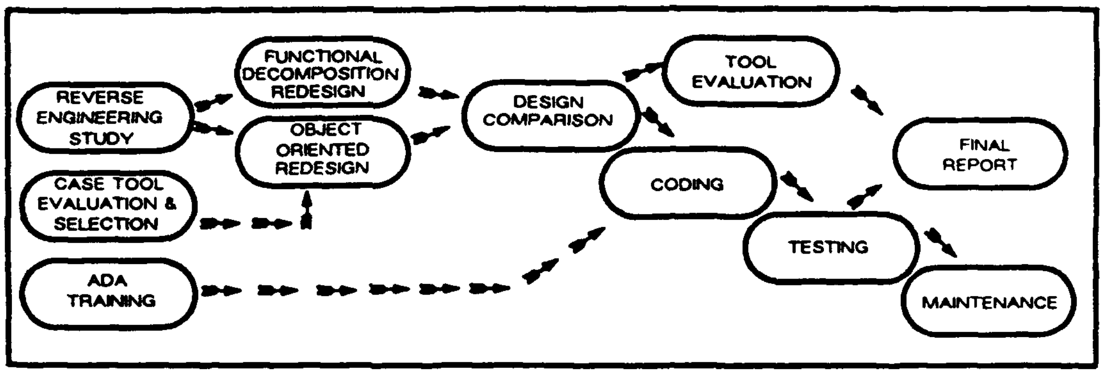
In this study we investigated several issues, methods, and tools that impact the software life cycle development that the Army uses to design, implement, test, and maintain application software. The purpose of this project was not only the evaluation of these methods but the identification of problems affecting the success of software development within the constraints of current development policy. This paper outlines a process for transitioning older systems using current software engineering methodologies.

The U.S. Army Information Systems Engineering Command (USAISEC) maintains over 100 Standard Army Management Information Systems (STAMISs). The need has been identified to modernize these systems. One way is to upgrade these systems from a COBOL, flat file, batch processing mode to systems written in Ada in order to increase functionality, maintainability, and reusability. While the appropriate technology exists for this effort, no systematic knowledge exists about how to effectively and efficiently make the transition. This lack of knowledge hinders the development of an appropriate transition strategy.

2.0 OBJECTIVE:

The objective of this project was to examine the application of various tools and methods to the transition of existing STAMIS applications from COBOL to systems written in Ada, designed for re-use, and utilizing newer technological capabilities. Specifically, this project examined the application of reverse engineering tools and methods, compared functional decomposition versus object-oriented design, determined specific CASE tool requirements for a STAMIS type environment, assessed Ada training provided by the Department of Defense, and established a framework for examining the maintainability/reusability of code.

Upon completion of the re-designed system, run time parameters and functionality will be compared to the existing system. The tasks completed under the project include: evaluation of CASE (Computer-aided Software Engineering) tools; reverse engineering a STAMIS to create a high level description of its functionality; redesign of the system using object-oriented and functional decomposition methods; and analysis/comparison of design approaches.



3.0 APPROACH:

AIRMICS (Army Institute for Research in Management Information, Communications, and Computer Sciences) in conjunction with SDC–A (Software Development Center–Atlanta) selected the Installation Material Condition Status Reporting System (IMCSRS), an operational STAMIS written in COBOL, to re–engineer and redesign using both functional decomposition and object–oriented design methods.

Two key issues for an initial transition effort include the selection of an appropriate system and the selection of appropriate personnel. Many development problems were avoided by selecting a system that was not extremely complex, involved few interfaces, had a reasonable number of lines of code, decent documentation, etc.. The knowledge and confidence acquired during the initial "Proof–of–Principle" effort can be used for future systems development.

3.1. Hardware Environment:

The hardware selection decision for this transition effort was based in part on the software tools selected - certain tools only run on certain platforms. The basic premise is - do not commit to a hardware platform until the software tools have been selected. Since the target language was Ada, we were not bound to a specific development platform. Because one of the strengths of Ada is portability, recompilation should be the only step required to port a system implemented in Ada to the target platform. Another consideration in the selection of hardware is that most CASE tools have extensive random access memory and secondary memory requirements. In addition, a multi–tasking windowed environment is normally required, especially for the more complex CASE tools.

The hardware chosen to implement the analysis and re-desiqn consisted of a SUN workstation. This machine was chosen not only for its compatibility with the operating environment already in place but also because it supports the graphical user interface (GUI) necessary for building the diagrams using the CASE tools.

3.2. Software Environment

3.2.1. Operating System

The system development was done under SUN OS 3.5 with the UNIX 4.2 Operating system. Again, this choice was based not only on compatibility but to increase the portability of the resulting code.

3.2.2. Programming Language

An Alsys Ada compiler was being obtained for use in the implementation stage of the project for coding the new system based on the designs.

Ada offers many features not found in other languages. It allows a program to easily interface the hardware. It also supports real time programming and current execution of several program tasks. These features are advantageous for embedded weapons systems. However, such features have limited value to an MIS system. The main drawback of using Ada for MIS system is its very awkward and complex input/output(I/O) routines. For example, in Pascal to print a line of text consisting of several different data types on the screen requires only a single statement. In Ada each type must have its own I/O package and separate statements are required for each. This makes programming more difficult and non-intuitive. For programs which are not involved in extensive I/O this is a minor inconvenience. However, for MIS systems where I/O is a major portion of the design, this drawback can become formidable.

3.2.2.1 Ada Training (Course Critique)

The following sections contain a course critique of Ada training taken by Major David Stevens of AIRMICS and Captain Richard Wassmuth of SDC-A in preparation for this design effort. This extract of their comments on the course is submitted as an assessment of the formalized Ada training being offered by the Department of Defense.

The four weeks of training were necessary to solidify the concepts of software design as well as the Ada language itself. The first week covered basic language as well as general software design concepts. The following two weeks were sufficient to learn the details of the language to include packaging, tasking and generics. The concepts were tested in an excellent group project assigned the last week. The amount of time allowed was appropriate to cover the different components of the Ada language.

The course assumes that the student knows about structured programming, data structures and their associated algorithms (stacks, queues, linked lists, etc.), and other software engineering principles. The course takes a "hands-on" approach to learning Ada with time equally divided between the classroom and the computer laboratory. The programming projects are designed to teach one or two features of the language at a time. This approach proved to be an excellent methodology for those migrating to Ada from C or Pascal. This was because they possess many of the same features. Those who had been programming in COBOL for many years had a difficult time grasping such simple concepts as passing parameters to a procedure or function. This made it even more difficult for them to understand the concept of tasks and object-oriented programming. Students that were not familiar with COBOL were for the most part able to master the new concepts without difficulty.

Lecture, followed by hands-on exercises to demonstrate the new concepts, was an excellent means of teaching the language. This teaching method has to be the most significant aspect of the class. While the students complete the exercises, the instructor is available to coach the student through any difficulties they would encounter. Two organizational changes could improve the overall training. The object-oriented design lessons were taught after the language concepts. Design techniques are the most basic concepts that must be learned to create maintainable software. Regardless of the language, software engineering techniques must be learned first. The student can then look to a specific language, Ada in this case, for constructs that will implement the design concepts. This means of learning, leads to the other change. The concepts of packages should be taught following the object-oriented design lessons. Packages are the most basic building blocks in the Ada language. They provide for the implementation of objects and collections of operations. The packages should be taught to be a critical part of the implementation, not just an optional construct.

The primary area of weakness observed in the class was the lack of their ability to decompose a problem. Most programmers in the class had a very difficult time in conceptualizing a real world problem and then translating it into functional modules that would make up the final program. This is clearly a major weakness in our programmers that can have a major impact on the maintenance of future systems. This would seem to imply that even a well designed system, can end up as a poorly structured program that is expensive to maintain.

It is recommended that in addition to Ada training the courses on problem solving be given. A beginner course in structured programming and algorithms should be a prerequisite to taking an Ada programming language course.

Overall, the training was excellent. However, Ada was designed to work from a high level of understanding of the problem to a more detailed level for implementation. Teach the language in the same manner. Work from a good design to the high level Ada constructs of packages, subprograms and tasks. Then teach the more low_level constructs of the language such as looping, assignments and data structures. Nothing is gained from a powerful language without a strong design.

3.2.3. CASE/Development Tools:

3.2.3.1. Overview of CASE (Computer-aided Software Engineering) Tools

Software Engineering is comprised of procedures (project tracking, project control, software quality assurance, configuration management, metrics), methods (analysis, design, code, test), and tools (coding, analysis, testing, project management, documentation). CASE includes various tools that can partially automate portions of the software development life cycle. Initially, the tools were primarily software aids to assist in the generation of graphical representations of a system (the representation form depends upon what method the specific tool supports). Current tools incorporate other features such as project management, requirements tracking, code generation, etc. CASE tools are often characterized as "Upper" or "Lower" CASE. Upper CASE tools support front-end developmental activities such as business modeling, requirements analysis, entity relationship modeling, etc., while lower CASE tools support back-end stages such as code restructuring and re-engineering.

Though various CASE tools perform well at various stages of system development, no true full life cycle tool exists at this time (though many claim to do so). In order for tools to be fully integrated, two actions need to be completed - a common repository; and translation mechanisms from stage to stage, e.g. translation of data flow diagrams to structure charts. National Institute of Standards and Technology (NIST) has published the Information Resource Dictionary Standard (IRDS) which outlines the standard manner in which to organize information within a repository as well as the operations to be supported on the repository. The key with the repository issue is that NIST has developed a standard which outlines the required interfaces to a repository. By vendors adhering to the IRDS standard, information exchange among various CASE tools is made possible.

3.2.3.2. CASE Evaluation Criteria:

Some of the CASE tool characteristics that were used in evaluating the various products that were under consideration for this design effort include:

functionality
interfaces
performance
time constraints
methods supported - data flow, process driven, data modeling, etc.
data dictionary
single, multi-user
code restructuring
requirements tracking (links features in the application to requirements identified in analysis and flags requirements for which no feature exists)
analysis
design
code generation
configuration management
error analysis
automatic documentation generation
cost

Because of the multitude of CASE products on the market (each with competitive strengths and weakness), the key in selection is to determine at the beginning of the project what characteristics are critical to the tasks to be accomplished (whether the project involves design of a new system, redesign, code maintenance, etc.). Once this is determined, the various tools can be compared against these critical success factors until the tool meeting most, if not all, factors is selected.

We selected the CASE tool suite produced by Cadre Technologies called Teamwork to be utilized in the redesign portion of the project (specific tools included - structured analysis, structured design, Ada structure graphs, and Ada Code Generator - as well as the required Teamwork environment module). The primary reasons these tools were selected was because they supported both object-oriented and functional decomposition design methods, generation of template Ada code, and support for 2167 documentation. This was also a fairly mature product, well supported, and one of the most popular CASE tools on the market.

The Teamwork CASE tools included modules for system analysis, information modeling, real-time system analysis, system design, and a utility called Teamwork/access which controls access to other software development packages, project management systems, and document production systems. All Teamwork components work, look, and feel similar. The user interface consisted of a mouse-driven windowing environment containing pop-up and pull-down menus. Multiple windows could be edited simultaneously.
All Teamwork products access a project database that collects project models and their

model objects (Structure Charts, Data Flow Diagrams, module specifications, etc.). This database creates a development environment to completely support the analysis and design phase of software life cycle development by furnishing a common interface to model objects.

Consideration was given to purchasing re-engineering/restructuring tools (e.g. Bachman toolset which is also good for data modeling), however, the decision was made not to due to the expense of such tools. If the system being re-designed was large and complex, such tools could play a critical role.

3.3. IMCSRS Redesign

One of the issues to be explored during this project was to examine which design method - object-oriented or functional decomposition - is the more appropriate for STAMIS-like systems. Object-oriented design is a fairly new and unproven method to design systems, yet guidance is to utilize this method for (re)design activities. Functional Decomposition is the more traditional and accepted design method. To explore this issue, project members were divided into two teams - one re-designed IMCSRS using the object-oriented approach and the other using Functional Decomposition. A comparison of the two designs is provided later in the paper, however, the general conclusion is that object-oriented, while being a more difficult method initially, produces a design that is better suited for re-use, maintainability, and simplification of complex systems.

3.3.1. Reverse Engineering

In the absence of an automated tool to perform an analysis of the source code, a task order was undertaken to investigate/develop a method to reverse engineer the IMCSRS systems. The findings of this task outlined a method for determining the original functional requirements of the system from existing code. This involved a bottom up analysis and transformation of source code coupled with a top down construction of the problem description.

3.3.1.1. Overview of Reverse Engineering Methodology

A method to reverse engineer a system is through the process of "synchronized refinement". Synchronized refinement relates the execution of the program with the program's functional requirements. This is done by simultaneously performing a bottom-up analysis of the source code with a top-down synthesis of the application model. Synchronized refinement should be applied only in situations where a detailed, line-by-line analysis of the source code is required. The analysis is controlled by the determination of design decisions.

Design decisions are data structures and coding practices intended to accomplish structural goals of the designer. When a program is written, the original designer makes a series of decisions to form a solution to a problem by decomposing it into pieces and then indicating how the pieces work together to solve the problem. The type of design decisions normally done during the reverse engineering process are the following:

1. Composition/decomposition - Programs are built up from separate modules. This is usually implemented in the code using separate procedures to divide the program into parts.

2. Encapsulation/interleaving - Sub-components interact with each other. If the interactions are limited and occur through explicit interfaces, the component is said to be encapsulated. If two or more components use the same section of code or the same data structure, they are considered to be interleaved.

3. Generalization/specialization - A program data structure performs a similar function or is coded similar to another component.

4. Data/Procedure - The designation of program variables and procedures is a design decision that may give information as to program function, if the designer chooses appropriate names for the application.

3.3.1.2. Project-specific Approach

One of the outputs this task was a high level description of the Installation Material Condition Status Reporting System (IMCSRS) - an operational STAMIS that we had selected for redesign. The current IMCSRS system is composed of approximately 10,000 lines of COBOL code and is batch oriented. This high level specification provided the starting point (an overall view of the system's functions) for a more detailed system description that could be used as the input to the redesign phase. The following steps outline the additional re-engineering steps taken to produce a detailed requirements definition:

(a) Before a system can be re-designed, the current system characteristics/functions must be well documented and understood. Because a functional description for the IMCSRS system had never been documented, this was the first task to be completed. Input into the creation of the functional description included system user manuals, programmers guide, maintenance guide, and relevant regulations governing the system, interviews with the system maintainer(s), and source code. A key to understanding a system's functioning is to concentrate on inputs and outputs. Most STAMISs are primarily concerned with data collection and the production of various reports. If intermediate or output information is derived from input data, the relevant code must be located and examined in detail.

Once the current system functioning was documented, the next step involved the production of a new functional description (requirements documentation) to include new functionality to be added to the re-designed system (e.g. interactive processing, interactive data validation, . . .). Once this has been accomplished, the functional description must be validated with the appropriate personnel to include the system maintainer, end-users, and the Functional Proponent. This functional decomposition walk-through is to ensure that no requirements have been missed.

During the re-engineering process it is important to not go into great detail with the existing code - if you do you will simply be re-implementing. The key is to discover the basic functions (as the system currently runs), then add new desired features (interactivity, etc.).

The methodology used relied on a variety of representation techniques . The were 4 main steps involved in reverse engineering IMCSRS:

1. Construct a textual description of the application. Concurrently devise and revise a system's requirements summary.

2. Construct nested Data Flow diagrams describing the relationships among the system's cycle, programs and major files.

3. Construct Jackson Diagrams describing the structure of the files used in the system.

4. Analyze the algorithmic structure of the system using the process of Synchronized Refinement.

Textual Description

The textual description was written to obtain an initial understanding of the application domain that the system models .

a. The description was derived from information external to the actual system source code. This was based on system documentation, (user guides, operating manuals) furnished with IMCSRS.

b. At the textual description level, references were not made to the programs's internal structure or to the programming language used in order to maintain a high-level of abstraction.

c. The process of deriving the description was repeated at the subprogram and program levels. Each process was described in terms of the requirements placed on it by the system.

d. A summary was produced that consolidated the overall functional system requirements in a list format from the textual description.

Data Flow Diagram

The next step in the methodology is the construction of Data Flow Diagrams (DFD). DFDs characterize the major functional activities and files contained in the system. Beginning with a Context Diagram that described the overall system behavior, multiple layers of DFDs were constructed in a nested, hierarchical structure.

Jackson Diagrams

After the Data Flow Diagrams were constructed, the input/output structure of each program was developed to give a more detailed description of the files. The technique used to represent this information is called a Jackson Diagram.

Note: The CASE tools used in this effort (IDE's *SOFTWARE THROUGH PICTURES*) provided an editor to construct Jackson Diagrams. The editor was integrated with a Data Dictionary and could perform consistency checks on the constructed diagrams.

Code Analysis

It was necessary to acquire a better understanding of the functions of IMCSRS. The documentation was at times ambiguous and the program had been modified using non-structured programming practices making the code hard to follow. The code had to be interpreted by performing a line-by-line analysis. This technique of synchronized refinement resulted in an annotated structural description of how the system functions.

Synchronized Refinement was accomplished to some extent by the use of the multiple-window environment provided on the SUN workstation. We designated two windows for the code analysis and application synthesis activities. Steps were eliminated by using a source code control system such as SCCS available under UNIX.

3.3.2. Object-Oriented Design (OOD)

3.3.2.1 Overview of OOD Methodology

Object-oriented design is a new approach to system modeling that evolved as a natural consequence to object-oriented programming. There is currently no universally agreed upon definition of what constitutes the object-oriented design approach. It has been described as a cognitive process for capturing, organizing, and communicating the essential knowledge of the system's "problem space" and for the model formed gives guidance on using specific techniques to map this "problem space" to a "solution space". The method focuses on the concept of abstraction - illuminating important system attributes/ entities/objects and suppressing other low level physical/functional implementation considerations. Terms such as information hiding, "black box" design, or encapsulation have also been used to describe this method because information concerning detailed design or implementation features need not be known by multiple developers implementing a system designed in this manner - only the interface specification need be known. With this method systems are decomposed based on the principle of hiding the design and implementation decisions about the abstractions. Only the interface need be known to instantiate an object or process within an object.

In the object-oriented environment, data are primary, procedures are secondary, functions are associated with related data, and a bottom-up approach is used for development. The major impedance to acceptance of this method is that it involves an unfamiliar way of thinking to most system designers (data versus functional perspective).

An object is defined as an entity (file, device, organizational unit, events, physical locations, etc.) that is itself defined by a set of common attributes and services or operations associated with it (sort, retrieve_value, read, write, etc.).

Perhaps the strongest features of this design approach include increased reusability, maintainability, understandability, and partitioning of the problem so that several programmers can implement the system concurrently and with little communication if the interfaces have been well defined.

The object-oriented design process involves the following phases:

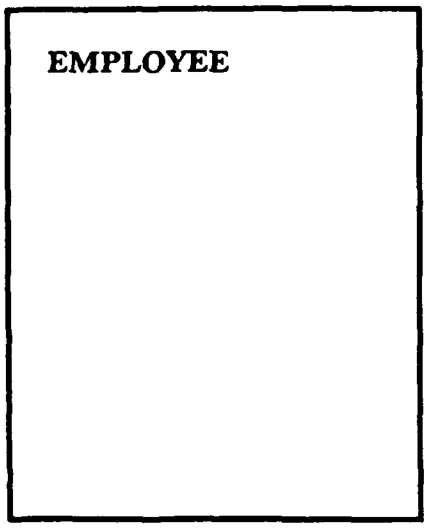
1. identify the objects and their attributes
2. identify the operations suffered by and required of each object
3. establish the visibility of each object in relation to other objects
4. establish the interface for each object
5. implement each object

The Object-oriented Design (OOD) methodology was studied in "Software Engineering with Ada", Grady Booch, Second Edition (1986). This book is an excellent guide to learning OOD and applying it to the Ada programming language. Note that the objective of the book is not to give the reader detailed instruction on the Ada language, rather a study of OOD and how it applies to Ada. It uses good examples, working from simple to more complex. Each example presenting a more detailed study of OOD and Ada. This book can be used as a self-study guide for even the beginning programmer. It was used as the "bible" for design of IMCSRS. The only shortcoming of the book is in its editing. Many typing errors can be found in the example Ada code causing some confusion and frustration to the reader.

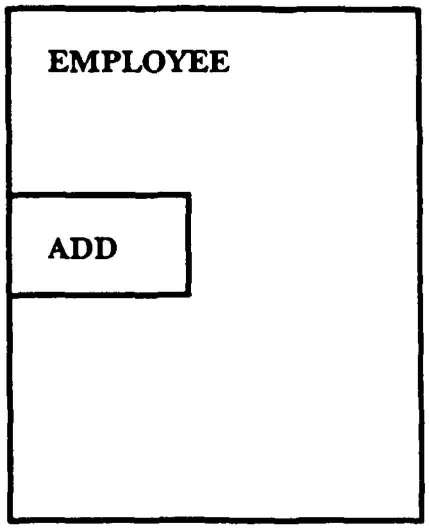
The object-oriented development method allows a designer to capture the "real world" problem domain in the software. The software and data structures can actually "look" like the problem domain being implemented. The steps currently being used for object-oriented design follows:

1. Identify the objects and their attributes.
2. Identify the operations that affect each object and the operations that each object must initiate.
3. Establish visibility of each object in relation to other objects.
4. Establish the interfaces of each object.
5. Implement each object.

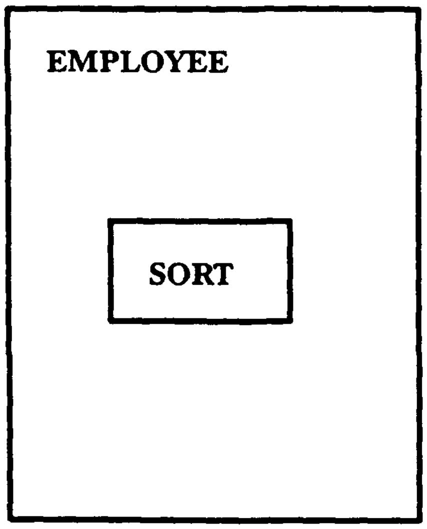
Identifying the objects is the most difficult step in the design methodology. A programmer can more easily associate data with an article, device, item, any world "thing." These objects can be a unit, device, telephone, gas, data base or any abstract object. Abstract objects, such as speed and time, do not have form. Each object has associated with it attributes that define the object. For the object HUMAN, some of the attributes are sex, height, weight and age. More appropriately, the attributes for EMPLOYEE could be social security number, employee number and salary. Buhr represents an object by a rectangle. The attributes are not graphically represented but are implemented as fields in the defining data structure.



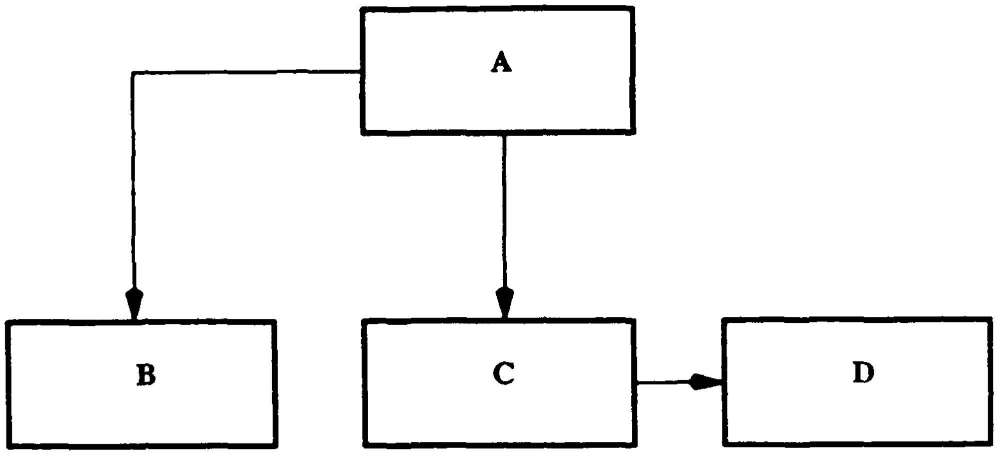
Each object has operations that act on it. Some of the operations are active, others passive. The active operations alter the object's state and the value of it's attributes. The passive operations do not alter the object but usually return the values of the object's attributes. The operations are established either as visible operations which are accessible by other objects or local operations that can only be used by the object itself. Each operation is displayed as a smaller rectangle inside the object. Buhr diagrams identify visible operations by "binding" them to the object. The binding is represented by attaching the operation to the side of the object.



Local operations are illustrated by separating the operation from the side of the object.



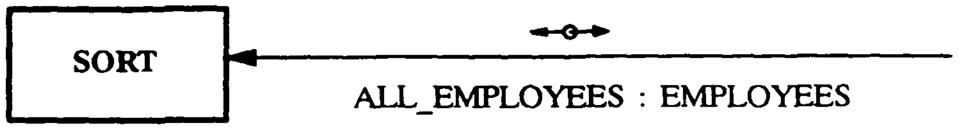
Once all of the objects, their attributes and operations have been defined, the visibility between objects must be established. The visibility will determine if one object "sees" or has access to the operations or attributes of another object. It is imperative for good software engineering to limit the visibility as much as possible. Buhr diagrams illustrate visibility by a directional arrow.

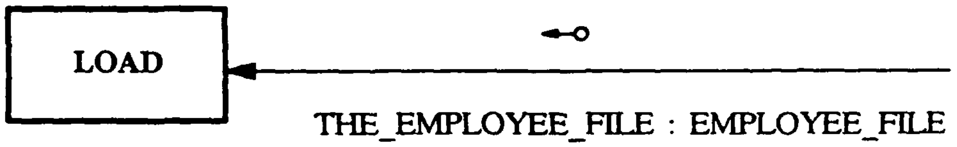


In this Buhr diagram, Objects B & C are visible to Object A. Object D is visible to Object C. Object A is not visible to any other object.

The next design step is to establish the interfaces between objects. The parameters passed between the initiating object and the receiving object must be established. The parameters are defined as **in** , **out** , and **in–out**. The **in** parameters are sent to the operation. The **out** parameters are received from the operation and the **in–out** parameters are sent in to the operation, modified and sent back out of the operation. The Buhr diagram illustrates these interfaces with a directional arrow, the parameter mode, formal parameter and parameter type.

In the following diagram, the **in–out** parameter ALL_EMPLOYEES of type EMPLOYEES is being passed to the SORT object for manipulation and return to the originating object after modification. The **in** parameter THE_EMPLOYEE_FILE of type EMPLOYEE_FILE is being passed to the LOAD object in order to initiate a process, in this case, the loading of the file to be manipulated. The contents of the parameter are not modified and are not returned to the originating object. The parameter attributes and types are stored in a data dictionary and must be consistent when interfacing objects (i.e. the type of parameter must agree between objects).



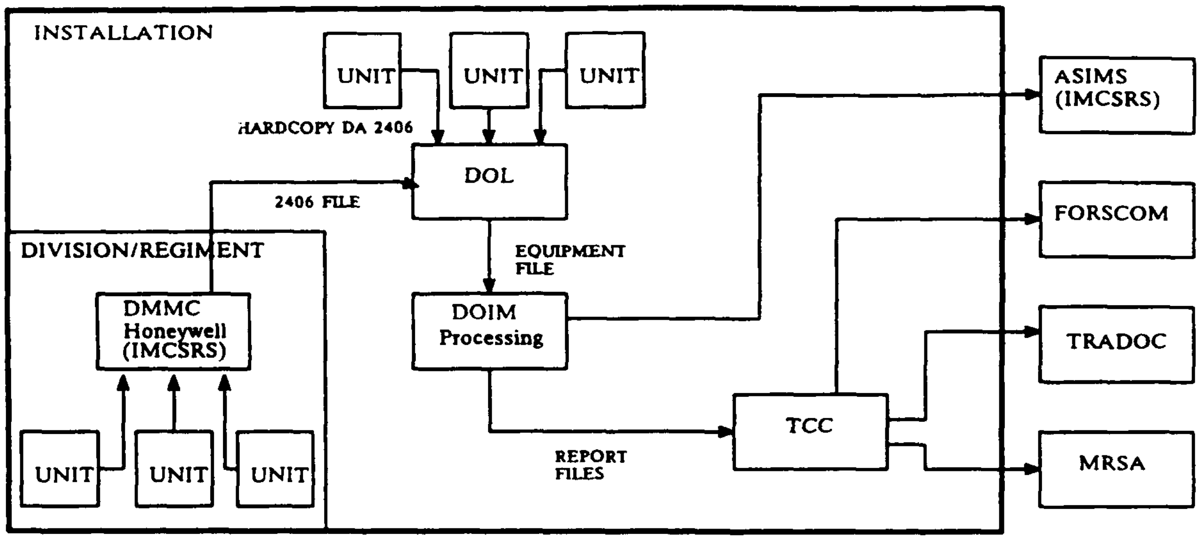


The final step in the design is to implement each object. The data structures representing the object will primarily be based on the object attributes. The bodies of the operations must then be developed.

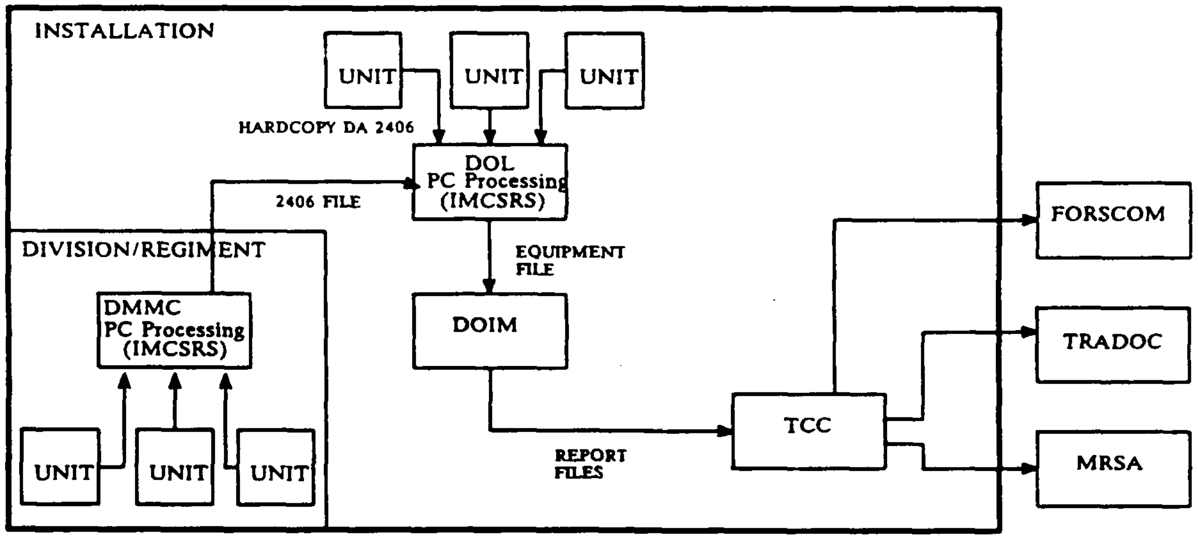
3.3.2.2 Project-specific Approach

The first step in the object-oriented design was to develop a functional description for the system. The requirements had to be defined as specifically as possible. The information was taken from the current COBOL system maintenance manual and available user guides. One of the most useful tools which helped with understanding the system, was the creation of a "System/User" chart. The chart showed the interface between the software and the user. The information on the chart included but was not restricted to the following

1. Software locations.
2. Hardware types. (mainframe, PC, etc).
3. Communications (Autodin, Modems, etc).
4. Input sources.
5. Report Destinations.
6. Overall responsibilities for the system.
7. Location of input/output processing.



OLD SYSTEM/USER CHART



NEW SYSTEM/USER CHART

The COBOL version of IMCSRS was used at the installation level, division level and regimental level. The system was processed on an Amdahl mainframe at the installation level and on a Honeywell mainframe at the division/regiment level. Trunks of the ASIMS (Army Standard Information Management Systems) network as and the Autodin network at the telecommunications centers (TCC) were used to transfer data to Forces Command (FORSCOM), Training and Doctrinal Command (TRADOC) and Materiel Readiness Support Activity (MRSA). Inputs came from reportable units in the form of hard copy DA Form 2406 and a pre-processed file of 2406 information from the sub-units. Additionally, a reportable equipment file was sent from FORSCOM via the ASIMS network to the DOL (Directorate of Logistics).

Reports went to the DOL who had overall responsibility for running the system. Because the system was operated on the Regional Data Center (RDC) the Director of Information Management (DOIM) at each installation actually input the data and received the output reports which were then delivered to the DOL.

After the functional description was validated for correctness, the system was designed using object-oriented procedures. One of the basic goals of object-oriented design is to mirror the real world. The System/User chart established a base-line to better understand the system and serve as a blueprint for developing the object-oriented design. Buhr diagrams were used to illustrate the design by CADRE'S "Teamwork" CASE Tool. Grady Booch's methodology was followed for the object-oriented design. The following steps were used to develop the object-oriented design of the system:

1. Developed a "System/User" chart.
2. Identified the objects.
3. Identified the attributes of each object.
4. Identified the operations that affect each object and the operations that each object must initiate.
5. Established visibility of each object in relation to other objects.
6. Established the interfaces of each object.
7. Implemented each object.

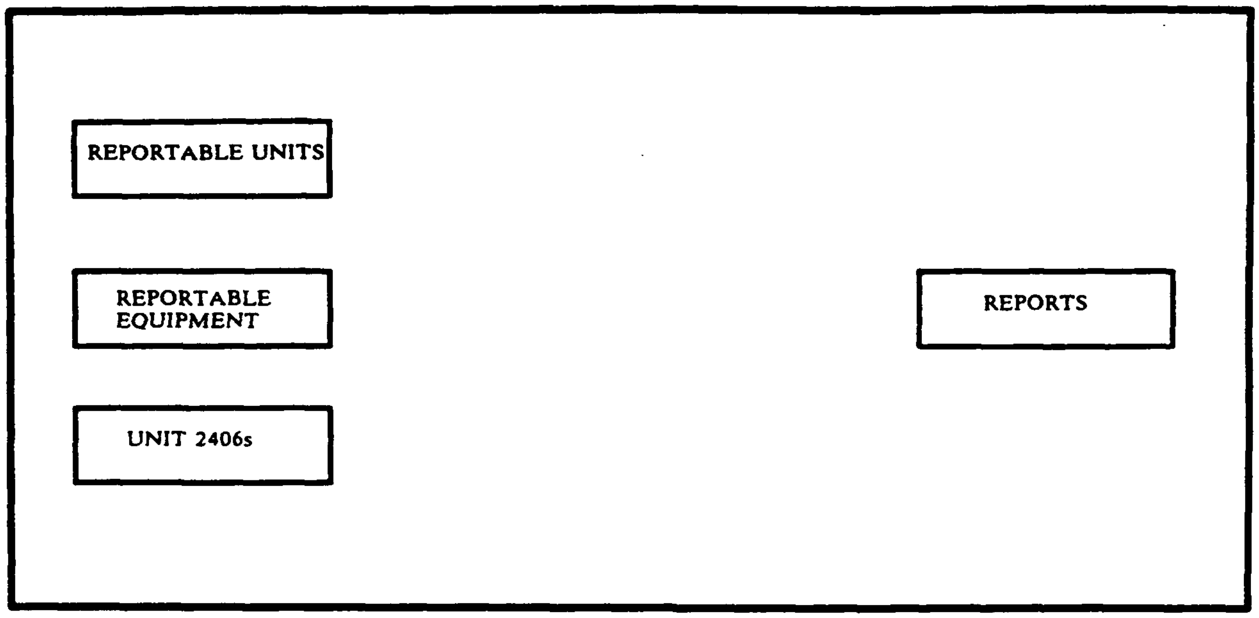
These steps were repeated many times during design refinement.

The new System/User chart was a modification of the mainframe version. The inputs and outputs remained the same but the processing requirements were relocated. The system was moved from a mainframe computer to a personal computer at the DOL. All inputs and outputs would be processed at the DOL. The DOIM would no longer be required to key the data to tape and run the batch system. Their only function would be to convert the desktop system output on disk to a format acceptable for transmission over the Autodin network.

Once the new System/User chart was developed, it was easy to identify the system objects. These high level "real world" objects were divided into four classes:

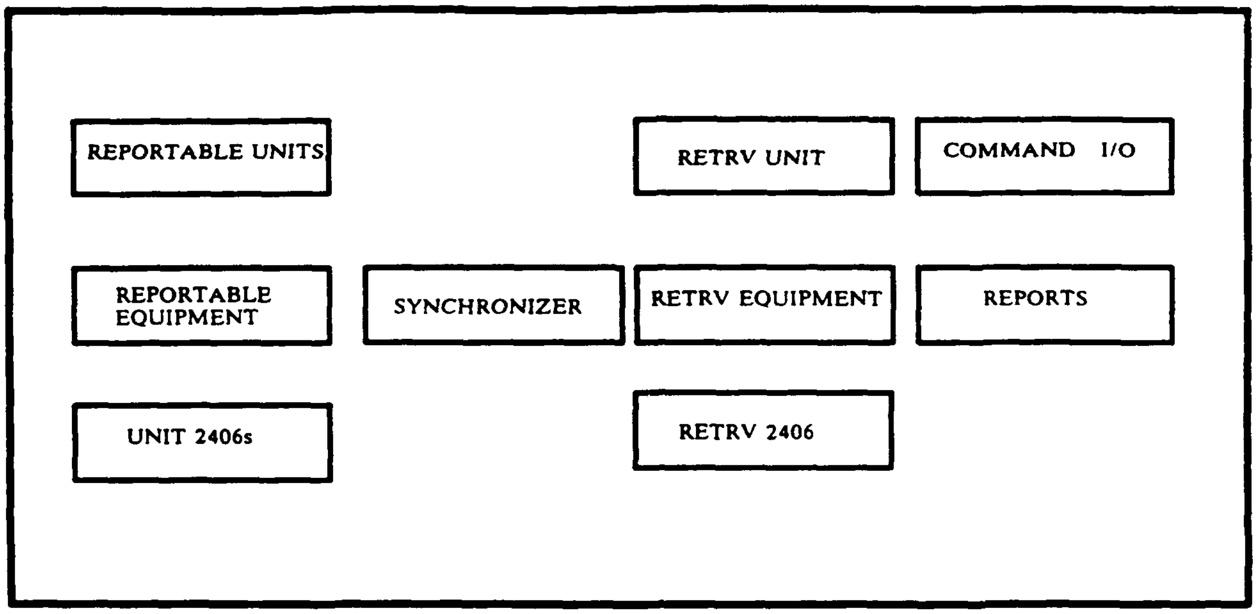
1. Reportable Units - the units that were required to report on the DA Form 2406.
2. Reportable Equipment - the equipment identified as reportable items.
3. Unit 2406s - the hard copy DA Form 2406.

4. Reports - hard copy and files generated by the system.



Other objects were defined during design refinement. These objects, often referred to as "abstract" objects, were used to supplement the "real-world" objects. Five abstract objects were defined:

1. Synchronizer - to manage the data bases.
2. Retrievers (3) - Retrieve Unit, Retrieve Equipment and Retrieve 2406 to give easy access to the data by the Reports object.
3. Command I/O - to control all high level menu display and selection operations.



Attributes were selected to define each object using the header information on the DA Form 2406. Other information needed to define the unit was subsequently added. Reportable Units were defined by the following:

1. Unit Identification Code (UIC)
2. Utilization Code
3. To Installation
4. From Unit
5. TOE_Number
6. Station/Division Code

Reportable Equipment attributes were limited to items defining a general piece of equipment. Only items of information that were common to all ECC/LIN and models of Army-wide equipment were included. Reportable Equipment was defined by the following:

1. ECC/LIN
2. Noun
3. Model
4. DA Organizational Readiness Rate Average

The Unit 2406 was defined by the fields on the actual DA Form 2406. The information was unique to the equipment posture of a specific unit. All general information was captured in the Reportable Units and Reportable Equipment objects. DA Form 2406 is divided into two parts, the header and equipment status (line items). The Unit 2406 object was divided in the same two parts to parallel the form. The outermost package defined the overall DA Form 2406 and included the following:

1. Period of Report (From)
2. Period of Report (To)
3. Date Prepared
4. Line Items

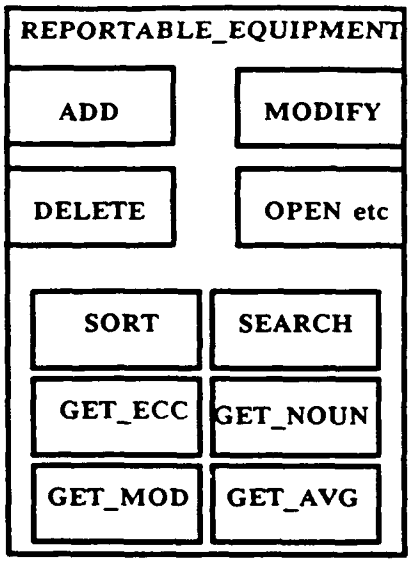
An object Line Item was derived as a smaller object of the object Unit 2406. It was defined by the following attributes:

1. Sequence Number
2. ECC/LIN
3. Model
4. Effect on System Code
5. Equipment Utilization Code
4. Authorized Quantity
5. On Hand Quantity
6. Possible Days
7. Available Days
8. Non-Available Days due to Organizational Supply
9. Non-Available Days due to Organizational Maintenance
10. Non-Available Days due to Support Supply
11. Non-Available Days due to Support Maintenance

During implementation the attributes became fields of the records representing these various objects.

Once each object was clearly defined, the operations that affected the object had to be identified. The basic operations were the same for each object. The Reportable Units, Reportable Equipment and Unit 2406s had to be Added to a list, Deleted from a list and Modified. The operations getting the information from the user for each attribute had to be defined (ie: Get_ECC, Get_NOUN etc). They also had to be Sorted and Searched. At a very detailed level, the implementation had to be considered. When each object was represented by a file more operations surfaced: Open, Close, Create, Save, Load and Compact.

Some of these operations were determined to be "visible" or available to an outside object; other operations would be used locally by the object itself. In the Buhr diagrams the visible operations were "bound" or touching the package (ADD, MODIFY), where the local operations were separate (SORT, SEARCH).



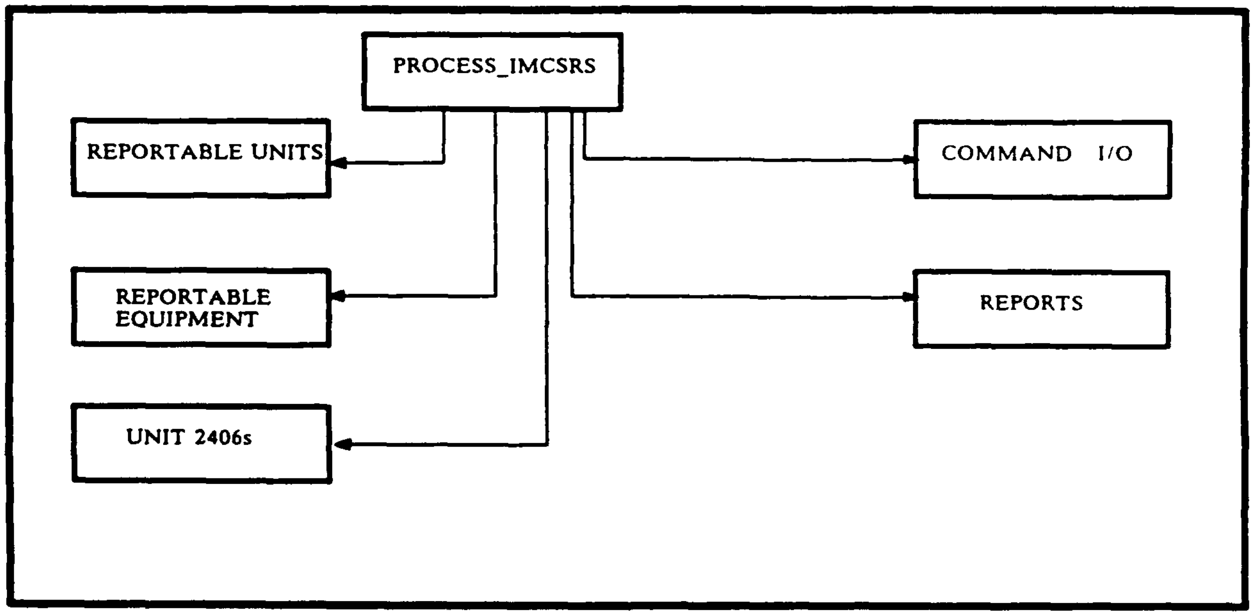
The "Retrievers" were defined to have one passive operation per object attribute. It was defined as passive since it had access to the data structure but could not alter it. The function would reach into the data base and retrieve the value of the specific attribute in the format defined. The data structure would remain unchanged. For example, Retrieve Equipment was defined to have the following "passive" operations (functions):

1. ECC_LIN - returns the current equipment ECC/LIN.
2. Noun - returns the current equipment name.
3. Model - returns the current equipment model.
4. DA OR Average - returns the current equipment DA operational readiness rate.
5. First Equipment - returns the position of the first piece of equipment in the list.
6. Last Equipment - returns the position of the last piece of equipment in the list.

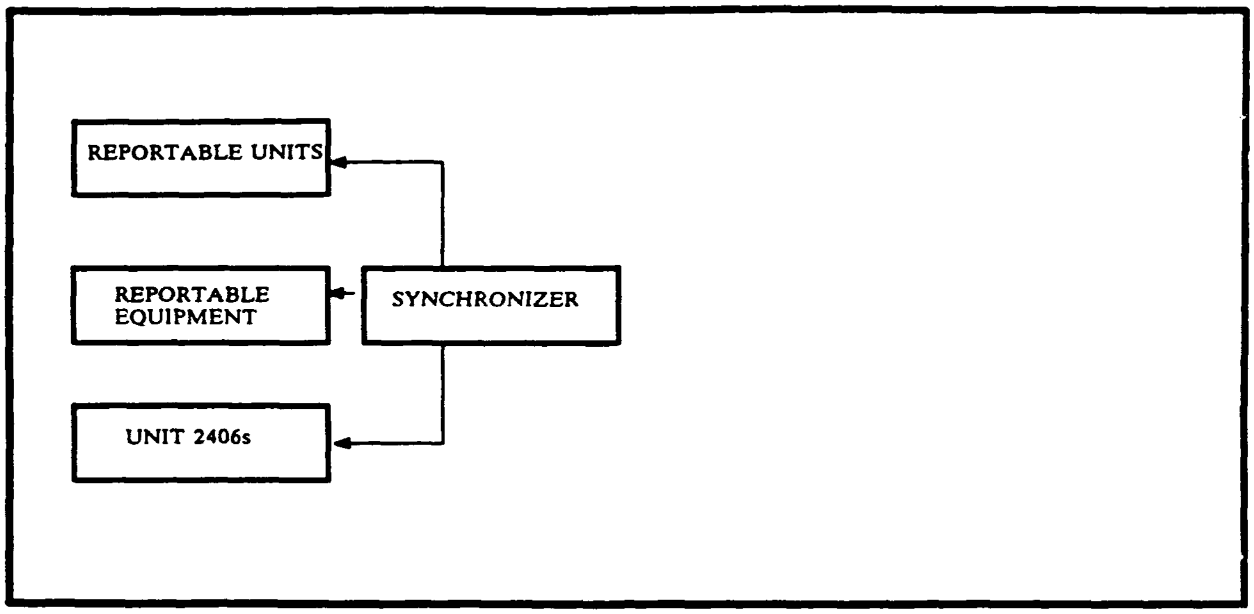
The Synchronizer was designed to control the data base for the Reports object. It performed nine operations.

1. Reset Units, Reset Equipment & Reset 2406 Items - moves to the first element in the list.
2. Next Unit, Next Equipment, Next 2406 Item - moves to the next element in the list.
3. Find Equipment, Find Unit - locates a particular element in its respective list.

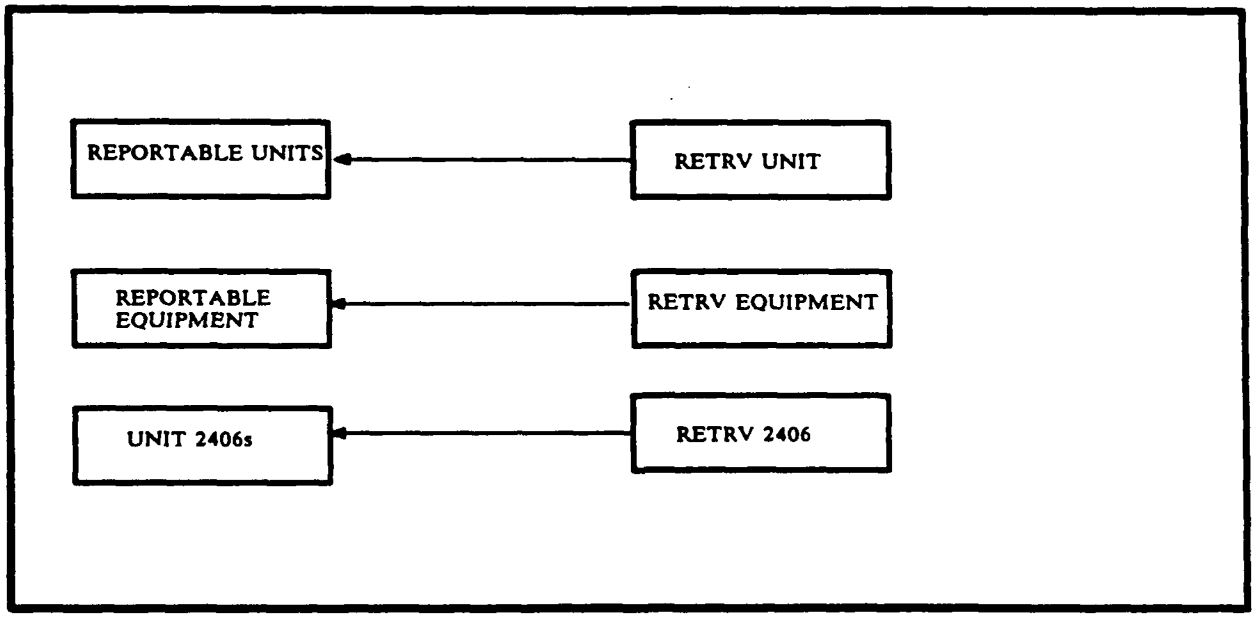
Minimizing the interfaces with other objects was a primary concern. Grouping the object with its operations met this goal; however, an object on its own serves no purpose-- another object must use the operations associated with it. For example, the Reports objects did not need to "see" the database. If the database was to be changed, the output reports would not have to be rewritten. That was the primary reason for the passive "retriever" packages. The "retrievers" would serve as an interface between the database and the report writers. Unless an object had a good reason for altering or even "seeing" another object, it was not given visibility. So the visibility sequencing started at the database.



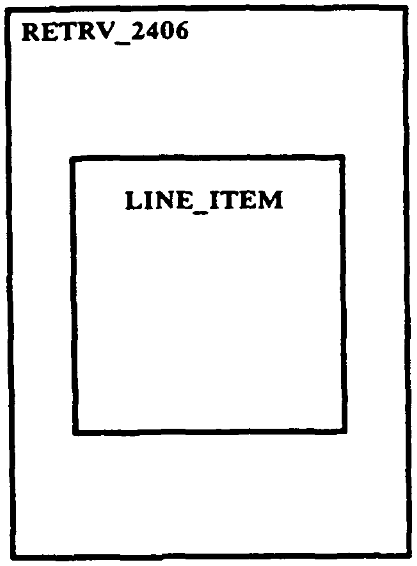
The main procedure, Process IMCSRS, was assigned visibility to all three data bases, Reportable Units, Reportable Equipment and Unit 2406s. This visibility was necessary to Add, Delete and Modify the specific objects within each class. The main procedure also was assigned visibility to Command I/O and Reports allowing control of the selected commands/menus as well as report generation.



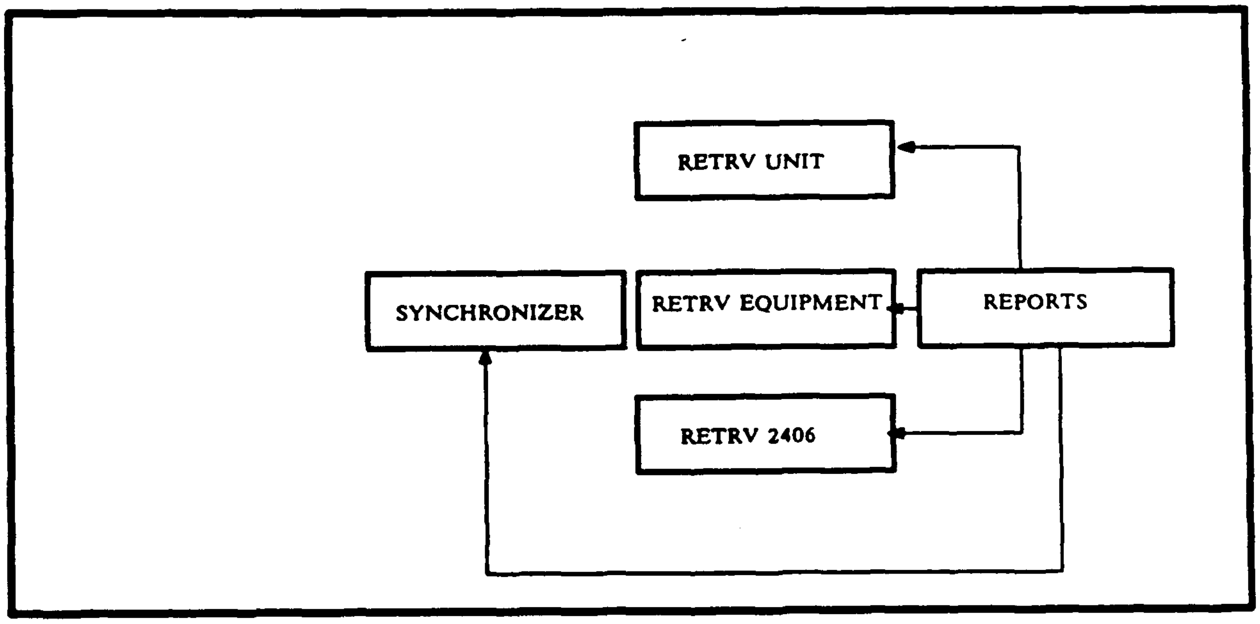
The Synchronizer was also given visibility to all the three data bases to ensure necessary data was pulled into memory from the files at the appropriate time. The Reportable Unit would be brought into memory along with its associated 2406. The first line item would be located on the 2406 and the Reportable Equipment information would be brought into memory. All necessary information would be in memory for use by the Reports object to write reports.



The "retrievers" were given visibility to their associated database only. Each attribute for that particular object would be accessible through the three retrievers. For example, Retrieve Unit had visibility only to Reportable Units.

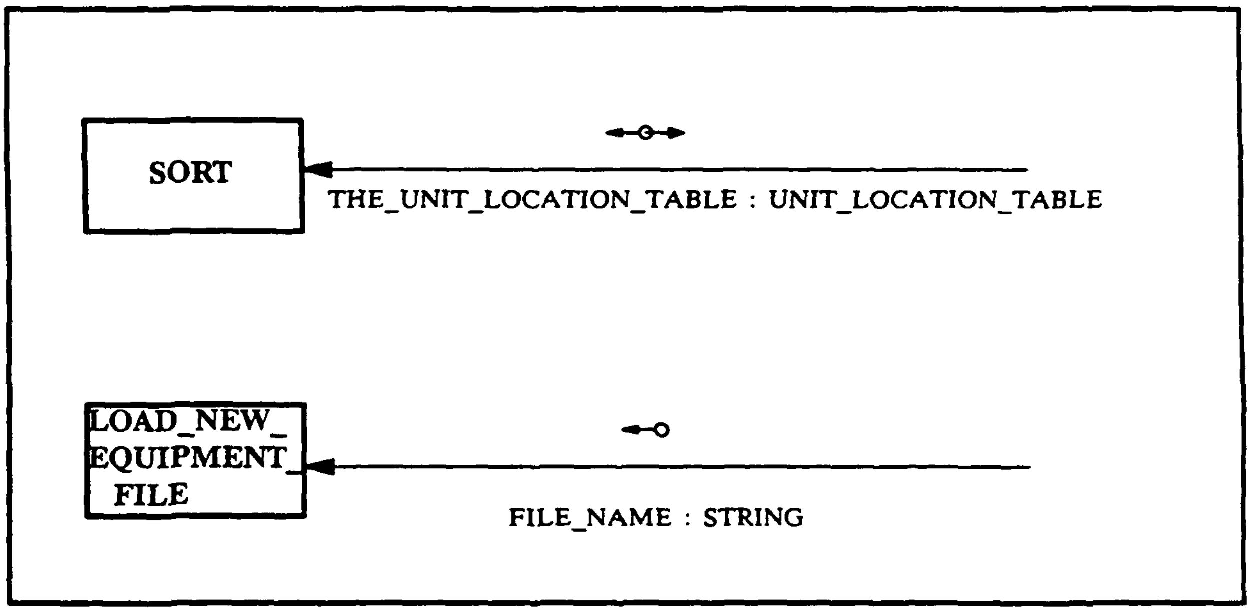


The RETRV 2406 was designed to look like DA Form 2406. The header information was contained in the basic package; the specific line item information was contained in the object Line Item.



Reports was given visibility to the "retrievers" and the Synchronizer. It had to have access to the data via the "retrievers" and control of selecting data from the three data bases via the synchronizer. The Reports object could then manipulate and use information from the data base without knowing how it was physically structured. By "hiding" the physical structure of the data base from the report writers, the data base could be removed and replaced in the future with a faster commercial data base management system without the reports being rewritten.

The visibilities were established to give specific objects access to operations and data of another object. The interfaces between the operations were established. The data that was sent and received between operations was identified and used in a subprogram formal parameter to establish the interface. For example, the SORT operation of Reportable Units would accept a table of type Unit Location Table, sort it in ascending order and return it. The couple (bi-directional arrow) displays an **in-out** parameter. If the arrow pointed toward the operation it would be an **in** parameter only, and inversely, if it pointed away from the operation, the parameter would be an **out** parameter only.



The high level components of the system were completed with the establishment of the object interfaces. The system had been separated into logical objects that reflected the real world. The operations required to operate on each of the objects had been identified and the visibility between objects and interfaces to allow the operations to be accessed had been established. The design was ready for implementation. To implement the design, only the data structures used to represent the objects and the bodies of each operation need to be written.

3.3.2.3. Results (Ada Structure Graphs, Ada Code)

The redesign resulted in three significant products. An up to date functional description was developed and used as a contract between the Proponent Agency and the designers. It specified exactly what the proponent wanted the system to do. The requirements definition is so critical during design that the functional description was mandatory.

Ada Structure Graphs (ASG) were generated by the CADRE "Teamwork" CASE tool. They implement "Buhr Diagrams" giving a high level view of the system. They are easy to modify for future software changes and should be an integral part of the programmers maintenance manual. The system is easily learned from these graphs. These diagrams can easily be translated into code specifications. The "invocations" or visibility lines generate the Ada "with" statements. The packages and subprograms are interpreted as the associated package, procedure, function and task subprograms. The invocations with coupling will show subprogram specifications with parameters. All design specifications to include the most detailed interfaces can be derived from these graphs.

The "Teamwork" CASE tool included Code Generator (CG) that interpreted the Ada Structure Graphs (ASG). The system read the Buhr diagrams and generated the system specifications. These system specifications were in usable Ada specification format. Body "stubs" were also generated for each package and subprogram. The system allowed notes which were inserted into the code as exceptions, data structures and program logic to be attached to each object . This code was then processed with a simple compiler that checked the design logic for completeness. It generated errors specifying the location of the incorrect design. After a clean compile, the Ada specifications and bodies were written to separate files for use during implementation.

3.3.3. Functional Decomposition (Structured Analysis and Design)

3.3.3.1 Overview of Structured Design Methodology

The method for redesign of IMCSRS from the functional decomposition side of the project consisted of a structured development methodology. In this approach, the system is analyzed from a functional perspective. The functions and procedural nature of the system were of primary interest; the data is considered during the later stages of development. The structured development is made up of two distinct phases: Structured Analysis and Structured Systems Design ("Top-down design"). Structured Analysis can be described as the activity of deriving a structural model and the accompanying structured specifications to give a clear "definition of the problem". Structured Design can be defined as the development of a computer system solution to a data processing problem, creating a new system containing the same components and process relationships outlined during the analysis of the system.

The particular structured development techniques used during the project were those outlined by DeMarco and Yourdon. There are slightly different approaches to structured analysis, but the DeMarco method contained the basic elements required. Also, the particular CASE tools chosen to implement the design and analysis of the system follow the concepts of DeMarco and Yourdon in their functionality.

The structured analysis of a system precedes its design; structured specifications act as direct input for the design phase. By developing a structured specification that is graphical using the CASE toolset, we obtain a model of the system that is very concise and easy to interpret.

The main building block of the Structured Analysis of a System is the Data Flow Diagram (DFD). A DFD is a graphical representation of a system that depicts the active processes and their interrelationships. DFDs consist of 4 basic elements: Data Flows (directional arrows) , Processes (circles) , Data Stores (2 parallel lines), and Sources/Sinks (rectangles).

Data Flows represent the movement of data. This is the symbol chosen to show the interface between components on a DFD. Data Flows are basically the media through which data are transmitted. They are labeled with names that described the composition of the data.

A process bubble is placed on a DFD when some transformation of data occurs. Data can be manipulated within a process either of 2 ways: (1) the structure of the data can change, producing as output a reformatted version of the original data (2) the information in the data can be transformed, generating new data. One of the most difficult tasks during the structured analysis phase is reducing system activities to processes that transform data. Each process should receive a descriptive name that gives a clear picture of the type of transformation occurring within the function.

A data store is where some temporary storage of data occurs. Data stores can represent files, information on magnetic tape, card indexes, entire databases, etc.

The Source/Sink symbol depict where the data required by the system originates and where the output of the system ends up. A source/sink can be a person or organization external to the system that is the net originator or recipient of data (a source is a data originator; a sink receives data). These symbols are only present on the context diagram for a system. A Context Diagram is a DFD at the highest-level of abstraction for a system; it is usually shown as a single process bubble containing the name of the system with data flows to and from sources and sinks. The context diagram should show the interface between the outside world and the system being developed.

The DFDs within the structured specification were developed using top-down partitioning. Top-down partitioning allows the system to be decomposed into a multidimensional arrangement with differing levels of detail so that there is a smooth progression from the most abstract (top) to the most detailed (bottom). By assigning functions in the system to unique elements within a DFD, redundancy is eliminated. The structured specification also has to be disassociated with the hardware, vendor, or operating procedures of the current environment. It needs to focus only on how the system functions. Unlike conventional flowcharts, DFDs don't show flow of control or procedure sequences. The system specification should only be concerned with the pieces of a system and how those pieces relate to one another, not the sequence in which they occur or the number of iterations involved. The main emphasis in a DFD is the flow of data. The structured specification is organized from the viewpoint of data, i.e. the processing that a piece of data goes through from beginning to end of the system. This gives a better overall view of the system since it is not limited to how a particular user interfaces with individual portions of a system.

In the structured design portion of a system development, the structured specification generated in the analysis phase is utilized as a roadmap towards the new design. By using the structured specification as a well-defined statement of the problem, structured design allows the form of the problem to guide the solution. The design itself is not the final system but a plan for implementing the new system. Structured design can also help outline a set of criteria for evaluating the proficiency of a design solution as it relates to the original problem. Structured design derives a simplified, graphical representation of a system by partitioning the functionality and organizing hierarchical relationships. Partitioning systems modularizes function; effectively isolating functions from updates and changes in different portions of a system.

The main tool within structured design is the structure chart. Like a DFD, the structure chart is a multidimensional arrangement of a system. A structure chart depicts the partitioning of a system into modules, the organization and hierarchy between the modules, communication interfaces between the modules, and the labels for the modules. The basic elements within a structure chart are: modules, invocations, and couples.

A module is represented by a rectangular box. The contents of a module (the module specifications) are not shown on the structure chart, but can be considered as a continuous set of program statements that have in common input/output, function, mechanics and internal data. A module could be a subroutine, a function, or system call. System or library modules are called pre-defined modules since their development is not part of the design; they perform a function that can be accessed without having to be coded. The modules on a structure chart serve the purpose of "black boxes" in that the actual implementation of the function are transparent to the designer. This allows the programmer flexibility in coding a particular function as long as the input/output of the module follows the design. The function within a module may be further broken down into sub-functions by invoking subordinate modules at a lower level of the hierarchy. This is in keeping with the structured design concept of "one function per module" but allows subtasks to execute specific portions of a function.

The connection between modules (invocation), depicted as an arrow from one module to another, shows module calls. Invocations do not show calling sequence for module communication nor does it show the number of iterations. It just shows that a module can potentially invoke a particular subordinate module. The details of module access are handled during actual implementation of a function.

The communication between modules is handled through coupling. Module invocations represent how modules are connected; couples show data items moving from one module to the next. Graphically, couples are drawn as an arrow with a circular tail. The amount of coupling between modules determines the measure of independence between them. The level of coupling can also be examined by the designer to ascertain the overall quality of a design. The higher the coupling (and therefore, the data dependence) between modules, the increase in the probability of a change in one modules, the increase in the probability of a change in one module affecting another module. Minimizing the amount of coupling minimizes the ripple effect of program changes and leads to a system that is more robust. Coupling within a system can be reduced by eliminating unnecessary relationships between modules.

Another method of designing a more robust system and ensuring proper partitioning of a structure chart is determining the level of cohesion within a module. Cohesion is examining how the activities within a single module relate to one another. Coupling and cohesion are interdependent in that the cohesion of a module directly affects the level of coupling between it and other modules. Constantine and Yourdon's methodology of structured design delineates 7 levels of cohesion possible between modules:

a) A functionally cohesive module limits itself to the performance of one and only one task. This level of cohesion within a structure chart ensures the highest level of maintainability within a system and is the most desirable level.

b) Sequential cohesion occurs when a module outputs information to be used as input in another module within a system. This type of cohesion is maintainable, but the amount of independence between modules is increased.

c) Communicatively cohesive modules contain elements that share input/output data from a common activity. Because of the interrelation between these modules that share access to global data, there is a tendency towards redundant coupling or duplication of function.

d) Procedural cohesion is distinguished by control flow between activities that are possibly unrelated and execute different functions with a module. This type of cohesion is characterized by flags or switches transmitted as data between modules.

e) Temporally cohesive modules are made of elements that are connected to each other only by a time sequence of events. These elements in the module do not necessarily share functionality. This type of cohesion can also lead to redundant coupling since there may be a tendency to reproduce code in other modules that have the same type of time dependence.

f) Logical cohesion describes modules that contribute to a similar category of functions to be executed external to the module. In this instance, a module is used to perform several functions depending upon how it is invoked. The activities within the module are forced to share a single interface.

g) The final type of cohesion, coincidental cohesion, is the least desirable in that it exhibits the worst level of coupling and maintainability. A coincidentally cohesive module performs activities that have very little relation to one another. They have no well-defined function; the calling module is often tasked with sending a control flag to the receiving module to decide what is to be done. This requires the calling module to be aware of the internals of the called module, which violates the "black box" concept.

The DFDs of the structured analysis phase are converted into structure charts in the design phase through transform analysis. Transform Analysis is a strategy for deriving a first-cut structure chart from the analysis of a system. The initial design is then taken through several iterations of refinement using the methods of determining cohesion and coupling. The main difference between a DFD and a structure chart is the hierarchical nature of a structure chart. Names of a resulting structure chart do not necessarily relate directly to names of processes on the original DFD. The beginning of the transform analysis process involves determining the portion of a DFD that contains the essential functions of a system. This central transform will outline the basic functions that must be accomplished within a system and lead directly to the development of a structure chart.

3.3.3.2 Project-specific Approach

The functional decomposition portion of the project was accomplished using the *Teamwork/SA* and *Teamwork/SD* CASE tools available from Cadre. Both of these tools were designed following the structured analysis and structured design methodologies previously mentioned.

Teamwork/SA is an environment for systems analyst enabling the creation and verification of functional systems specifications. *Teamwork/SA* consists of tools for the rapid creation and editing of DFDs, process specifications, and data dictionary entries. Built-in Model Configuration Management tools organized models into leveled sets and helped establish relationships between DFDs and their corresponding process specifications. A versioning facility provided an audit trail during project development by tracking the history of project updates for review. *Teamwork/SA* also maintains a consistency check function with detailed knowledge of structured analysis that gauges system quality for correctness.

Teamwork/SD is an integrated set of design aids for employing a structured systems design methodology. This tool for creating structure charts, module specifications, and other graphic symbols contained within the design process provided a rapid graphical user interface for planning a system. There was also an embedded design rule checker in *Teamwork/SD* that would verify a proposed design's overall validity.

The CASE tools were relatively easy to become familiar with, since there was a standard interface and access method for the Cadre products. There was also tutorial information for the *Teamwork/SA* tool that familiarizes the user with the various editors and object creation tools.

Learning the structured design/analysis methodology was a matter of reviewing several books, articles, and technical reports on the subject. The choice of DeMarco and Yourdon was due to the connection between their particular implementation of the design process and the CASE tools chosen to develop them. Since the project was involved with the transitioning from an existing system to a new design, parts of the normal life cycle development for a system were either merged or excluded from the actual steps in the design. The lack of functional users and access to up-to-date documentation on the existing system made it necessary to make certain assumptions about functionality within the system. It was not clear initially whether the system was to be designed as a more structured version of the old system or if a completely new design with more efficient functionality using better file access methods was the goal. Originally we approached the design with the former goal in mind, therefore relying heavily on the description of the old IMCSRS system. Once the choice was made to implement a new design of the system, the old environment was used only as a foundation.

3.3.3.2.1 Systems Analysis

The initial problem in developing a new design for IMCSRS using functional decomposition was to analyze the functional requirements of the new system. The data flow diagrams (DFDs) developed during the reverse-engineering of IMCSRS were helpful in understanding how the processes within the current COBOL version of IMCSRS interacted, but if we wished to take advantage of a structured methodology within the new system, it would be necessary to come up with DFDs for the new design.

We needed to begin our analysis by determining the overall context of the new system. The context of the system basically describes how the system interfaces with the real world; it outlines the boundary between the outside world and the internal functions of the system. The context diagram for the system is a very simple diagram depicting the external processes that interact with IMCSRS as sources/sinks. On this diagram, we are only concerned with 3 interfaces: 1) Operating system, 2) Data entry operator, and 3) Reporting units.

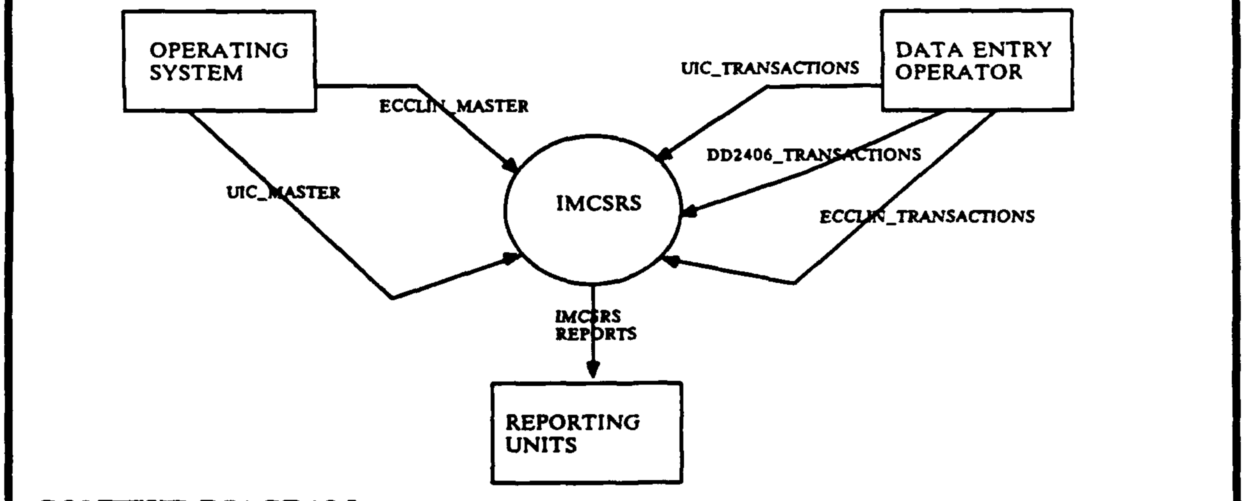
-- The operating system communicates with IMCSRS by not only controlling the program execution, but also by supplying the UIC_MASTER and ECCLIN_MASTER files as input.

These files are updated by IMCSRS.

-- The data entry operator is the person or persons within the DOL (Directorate of Logistics) that supply the information from the DD 2406 used to generate the 2406 transaction file. On the current version of IMCSRS the information exists as card input that the data operators run through the system in batch mode. In the new system, the 2406 will be entered through a menu-driven screen interface. The operator also enters any requests for updating the ECCLIN_MASTER and UIC_MASTER files

-- The reporting units include the installation, division, and major command recipients of the output reports. MRSA, FORSCOM, and TRANSCOM are part of the group, receiving their information via AUTODIN.

How the sources/sinks receive the output data or produce the input data is not important at this level of abstraction during the analysis of the system. They only appear on the context diagram to represent the external view of IMCSRS.



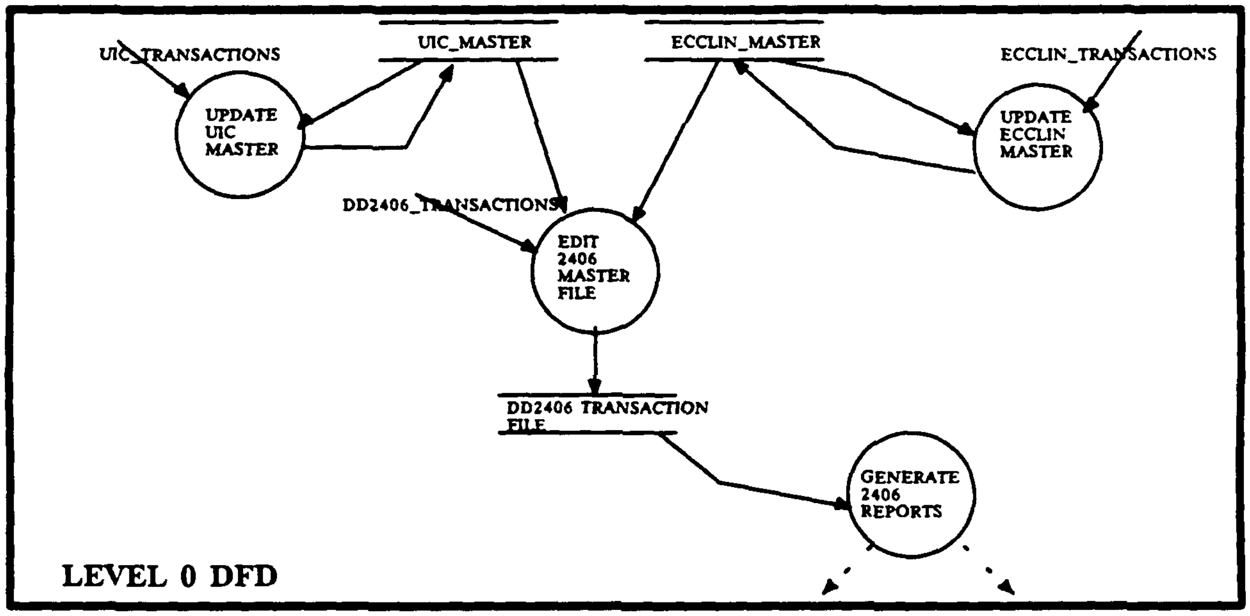
CONTEXT DIAGRAM

The next level within the DFD hierarchy, level 0, outlines the major processes that take place within the system. IMCSRS performs 4 different functions:

1. Updates the UIC_MASTER file
2. Updates the ECCLIN_MASTER file

3. Creates, edits the 2406_TRANSACTION file
4. Generates output reports

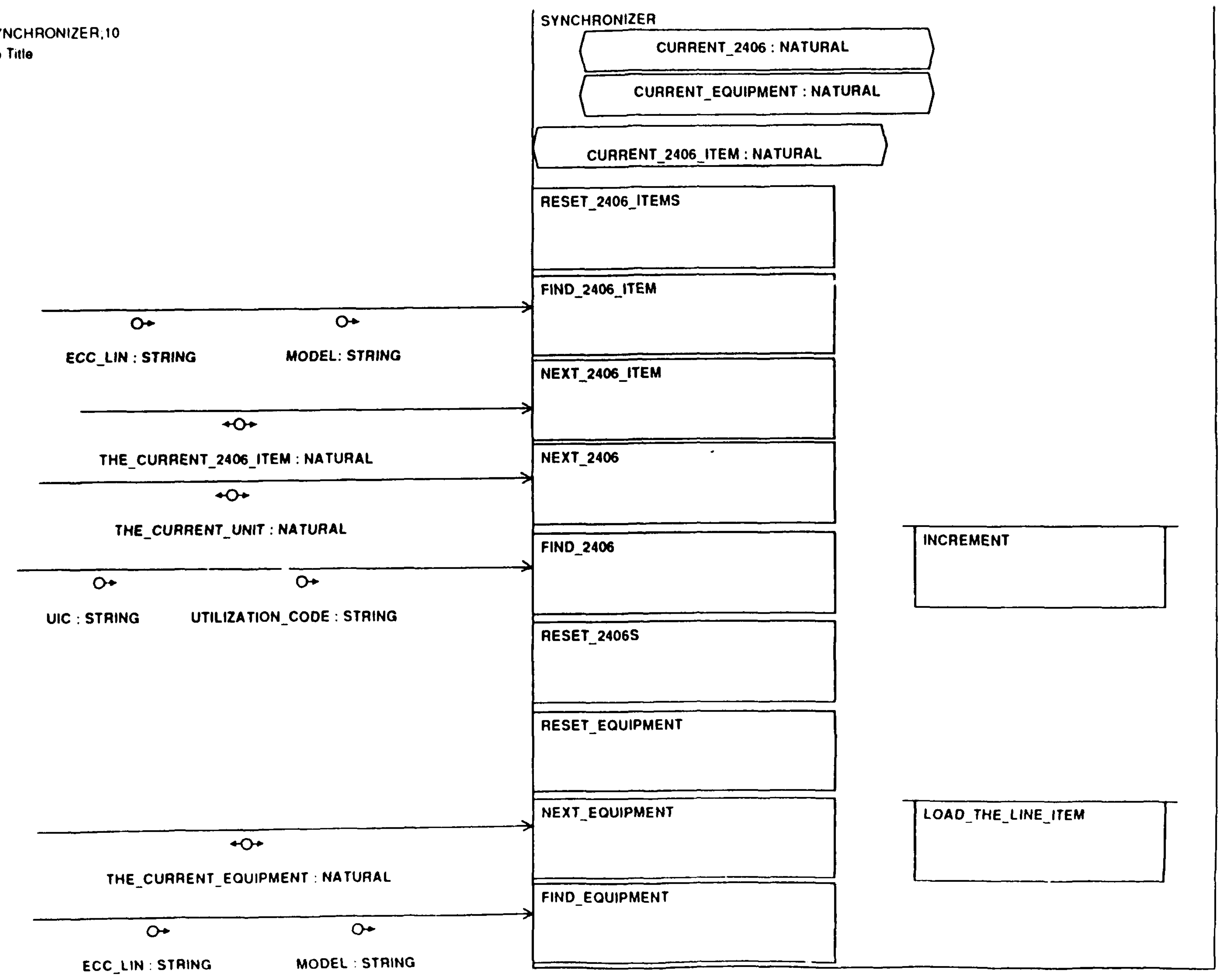
The process bubbles were labeled accordingly. Data flows that enter a process bubble without an originating process are considered to enter the diagram from one level higher in the hierarchy. For example, the UIC_TRANSACTIONS data flow is a component of information obtained from the data entry operator at the context diagram level. By the same token, data flows that have no termination process extend to the parent DFD. The child DFD is a breakdown of the inputs/outputs from its corresponding parent process. All data flows leading away from the GENERATE_2406_REPORTS process bubble would appear in a data dictionary as components of the IMCSRS_REPORTS data flow entering the REPORTING UNITS sink on the context diagram.



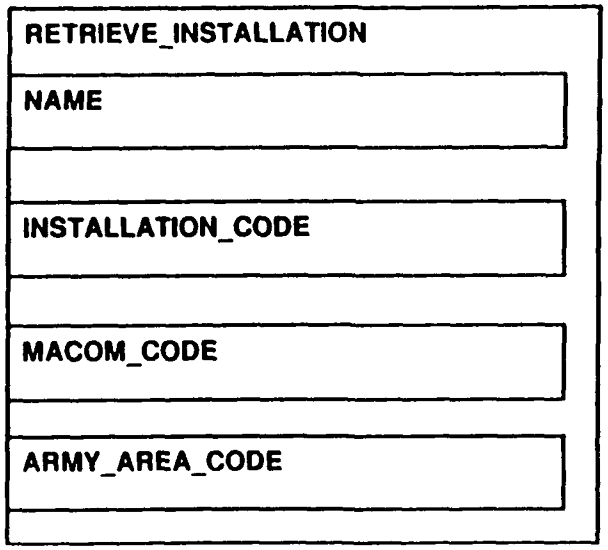
The files that are being used at this level are represented as data stores. When a data flow enters or leaves a data store and is not labeled, all the data items contained in the data store are being passed as one unit. The data flow leading from the UIC_MASTER data store to the UPDATE_UIC_MASTER passes the entire UIC_MASTER_RECORD for processing. The non-labeled incoming arrow for the UIC_MASTER store indicates that the entire updated record is being sent to the master file for storage.

So far, we are still only identifying the processes that take place within IMCSRS, not how they execute their functions. As an illustration of how the decomposition of the processes takes place, look at the level 0 DFD. Process bubble 1, UPDATE_UIC_MASTER, is not

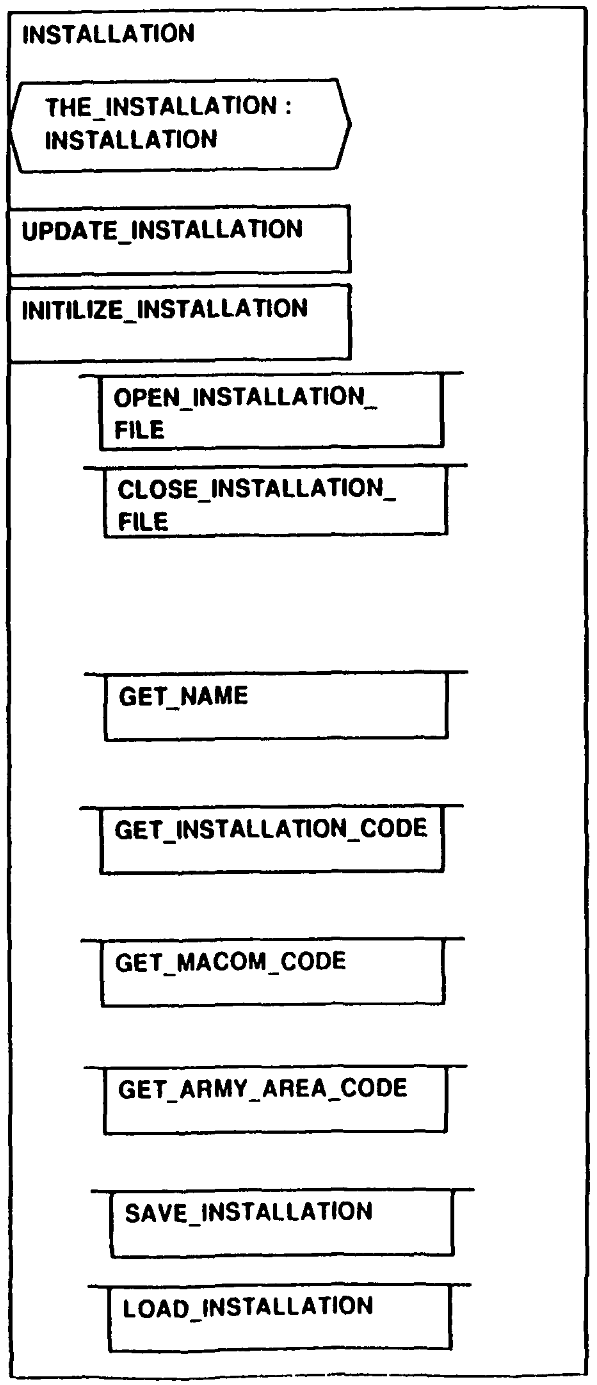
SYNCHRONIZER.10
No Title



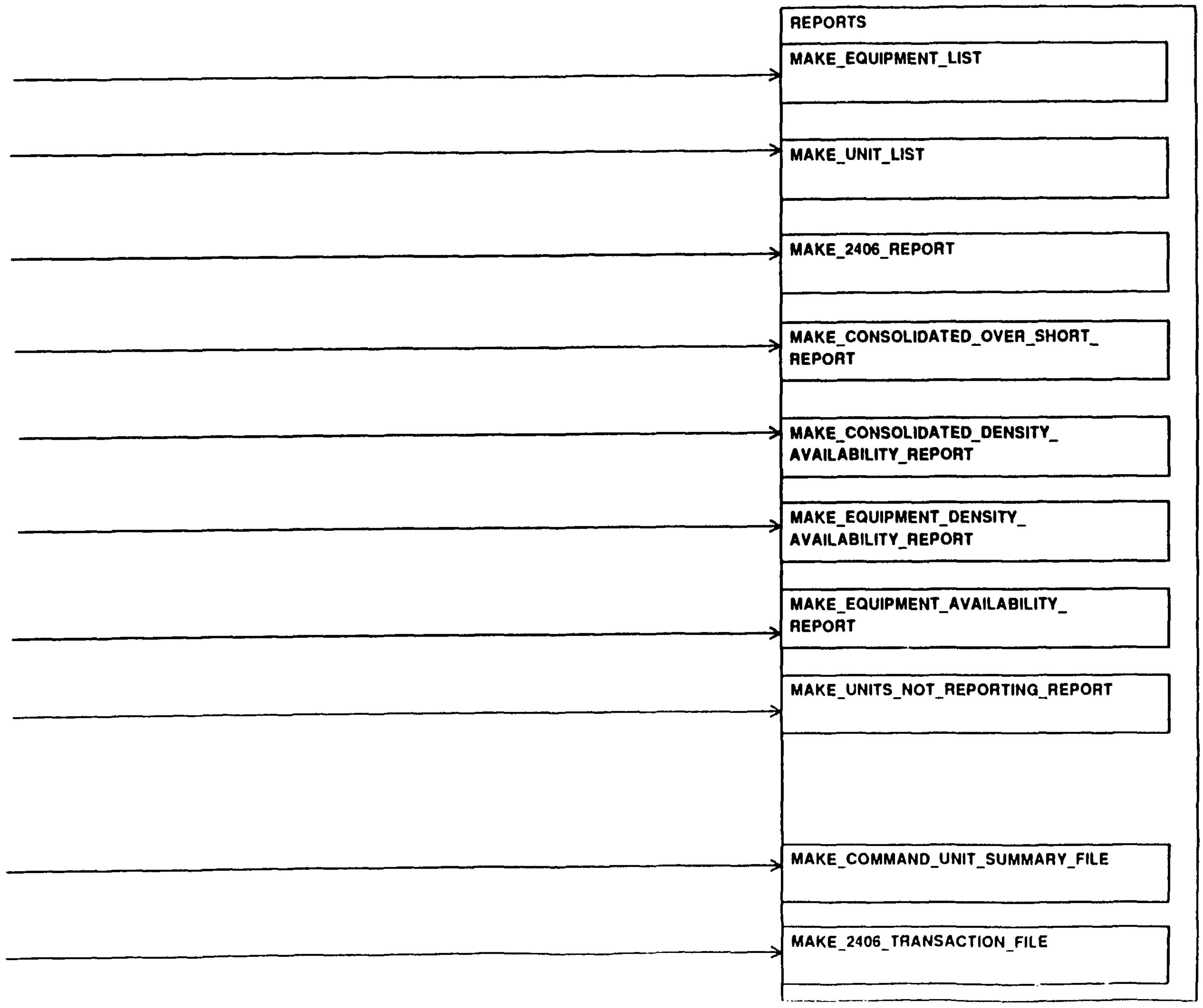
RETRIEVE_INSTALLATION;1
No Title



INSTALLATION:4
No Title

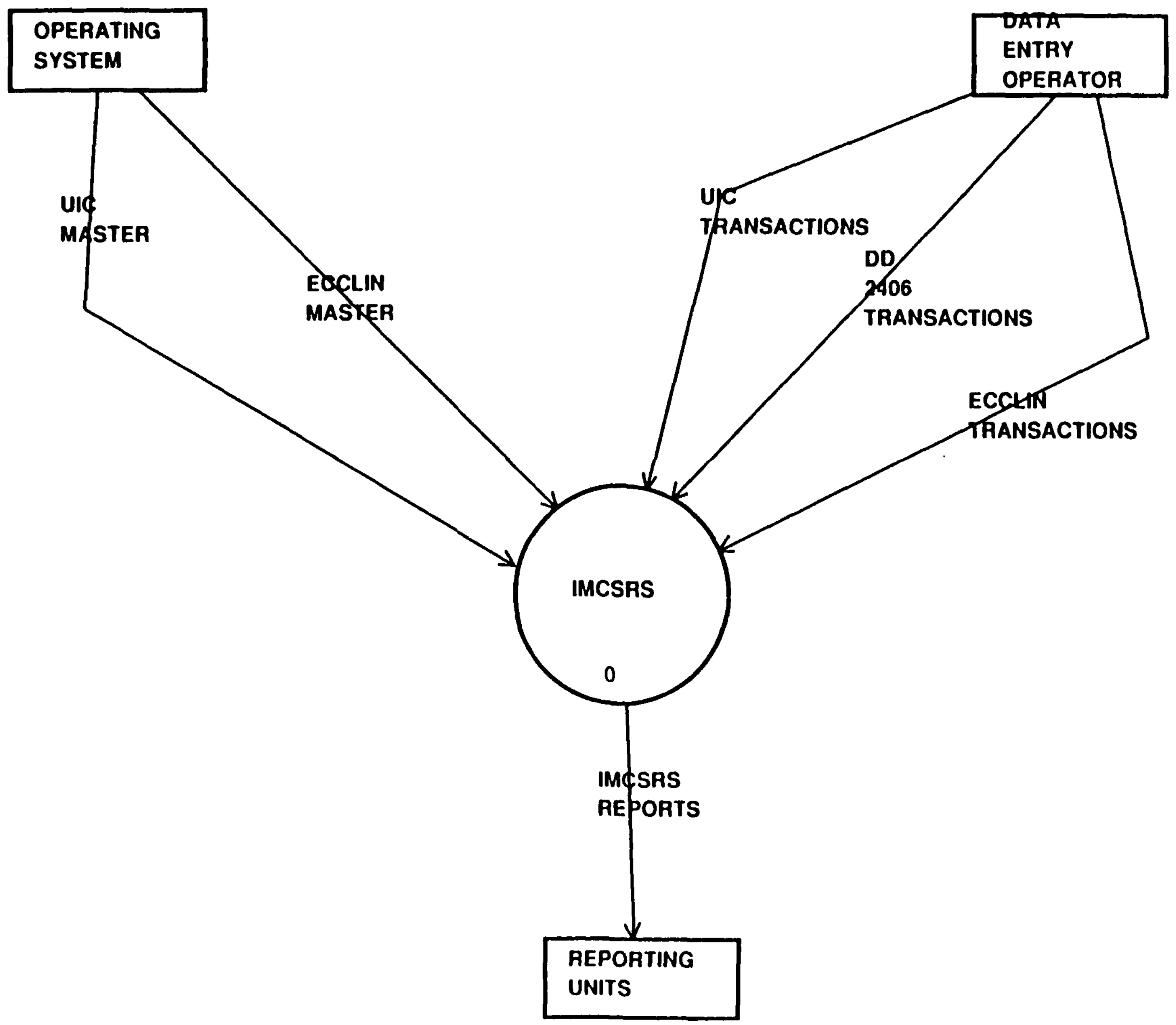


REPORTS;23
No Title

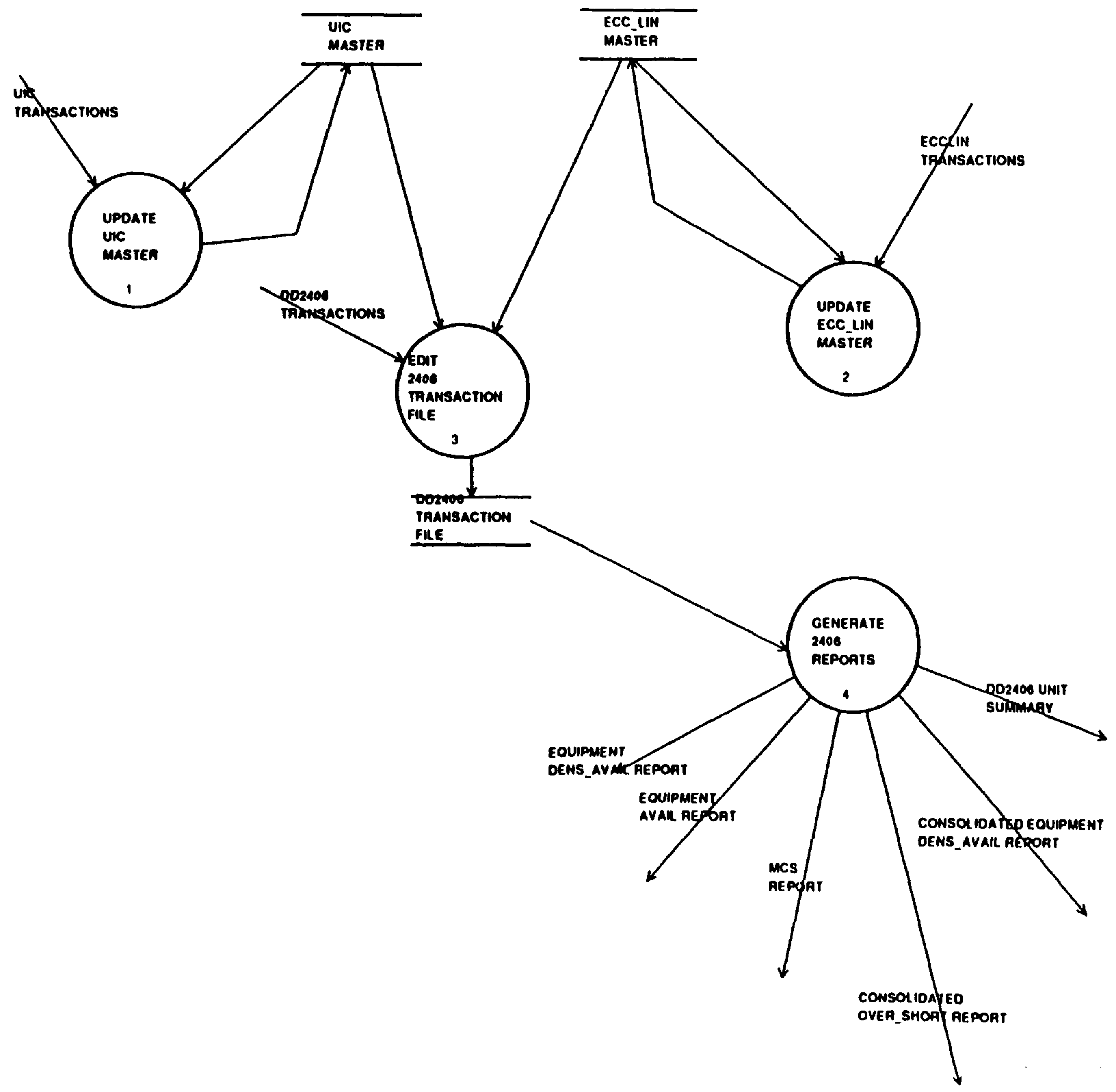

REPORTS
MAKE_EQUIPMENT_LIST
MAKE_UNIT_LIST
MAKE_2406_REPORT
MAKE_CONSOLIDATED_OVER_SHORT_REPORT
MAKE_CONSOLIDATED_DENSITY_AVAILABILITY_REPORT
MAKE_EQUIPMENT_DENSITY_AVAILABILITY_REPORT
MAKE_EQUIPMENT_AVAILABILITY_REPORT
MAKE_UNITS_NOT_REPORTING_REPORT
MAKE_COMMAND_UNIT_SUMMARY_FILE
MAKE_2406_TRANSACTION_FILE

APPENDIX B (Data Flow Diagrams, Structure Charts)

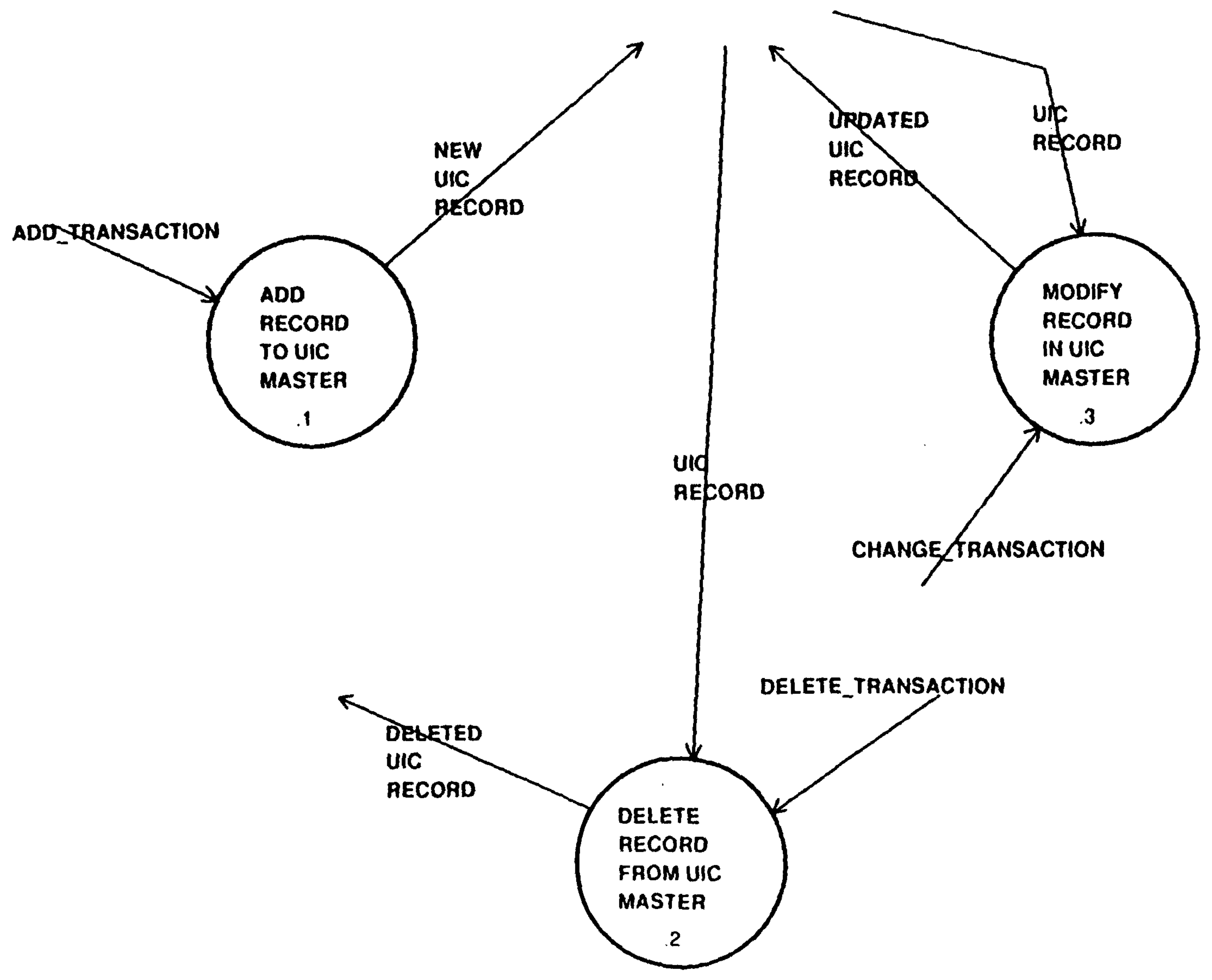
Context-Diagram;7
IMCSRS CONTEXT DIAGRAM



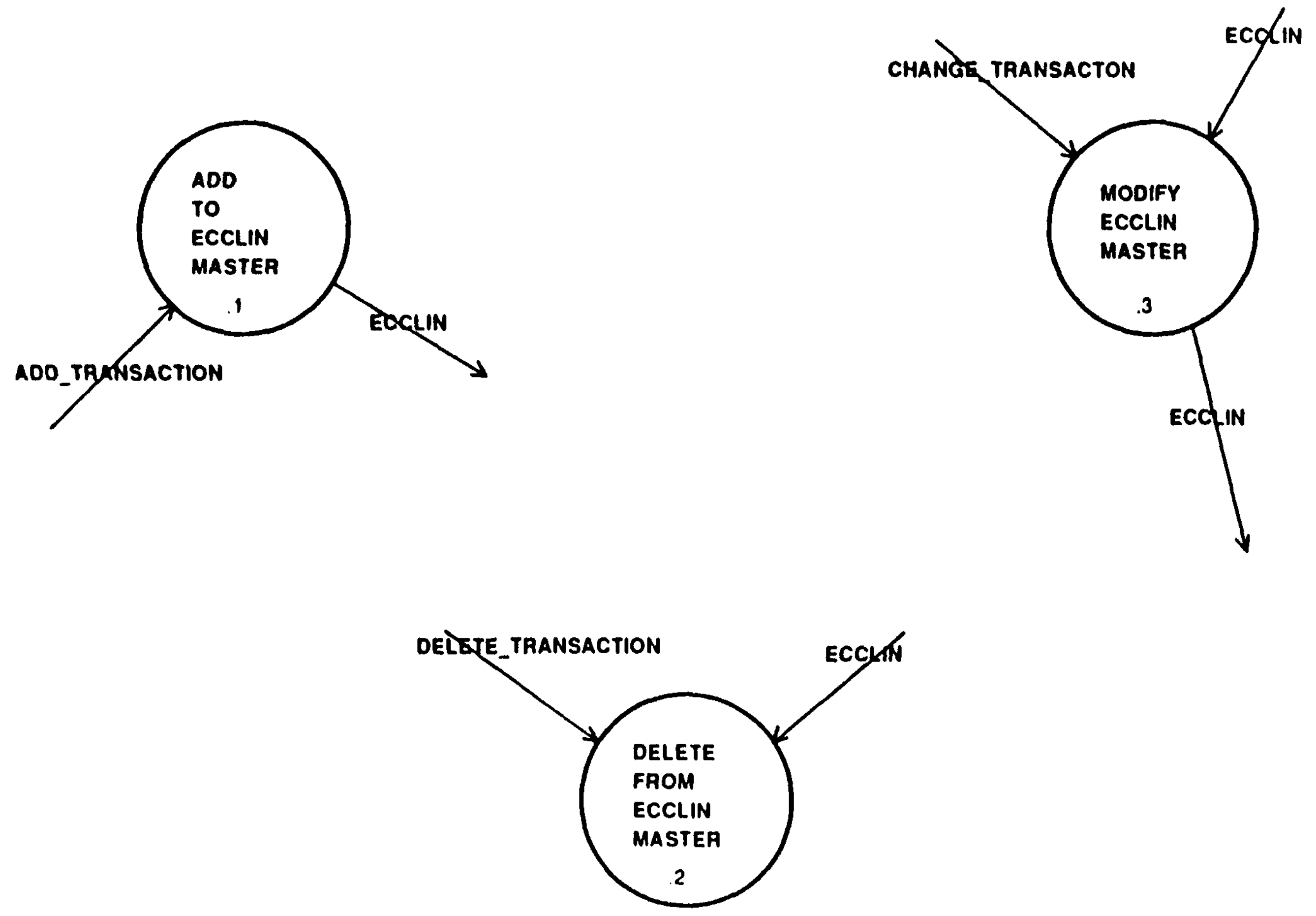
0.10
IMCSRS


UIC
MASTER
ECC_LIN
MASTER
UIC
TRANSACTIONS
UPDATE
UIC
MASTER
1
ECCLIN
TRANSACTIONS
DD2406
TRANSACTIONS
EDIT
2406
TRANSACTION
FILE
3
UPDATE
ECC_LIN
MASTER
2
DD2406
TRANSACTION
FILE
GENERATE
2406
REPORTS
4
DD2406 UNIT
SUMMARY
EQUIPMENT
DENS_AVAIL REPORT
EQUIPMENT
AVAIL REPORT
CONSOLIDATED EQUIPMENT
DENS_AVAIL REPORT
MCS
REPORT
CONSOLIDATED
OVER_SHORT REPORT

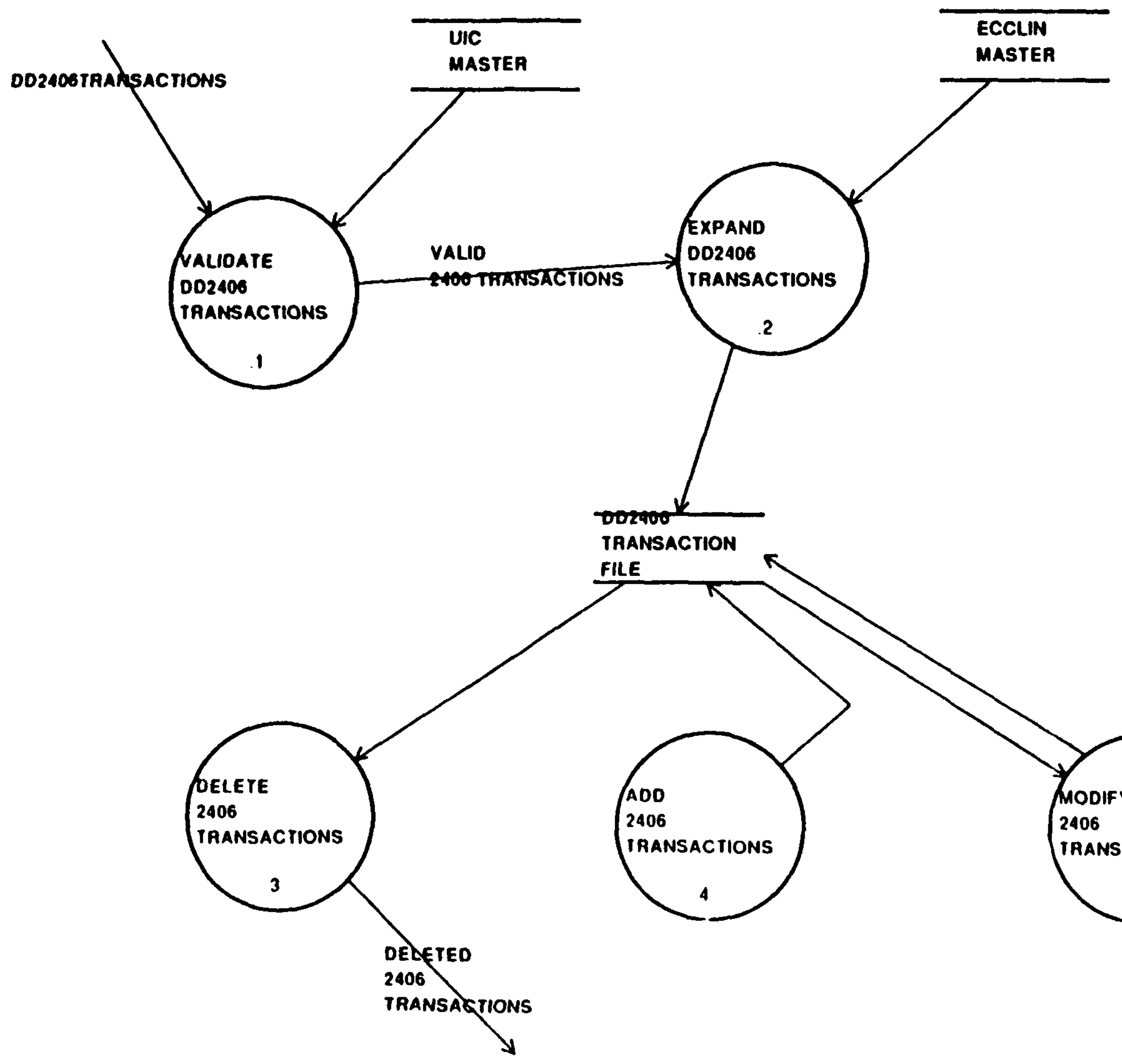
1.2
UPDATE UIC MASTER



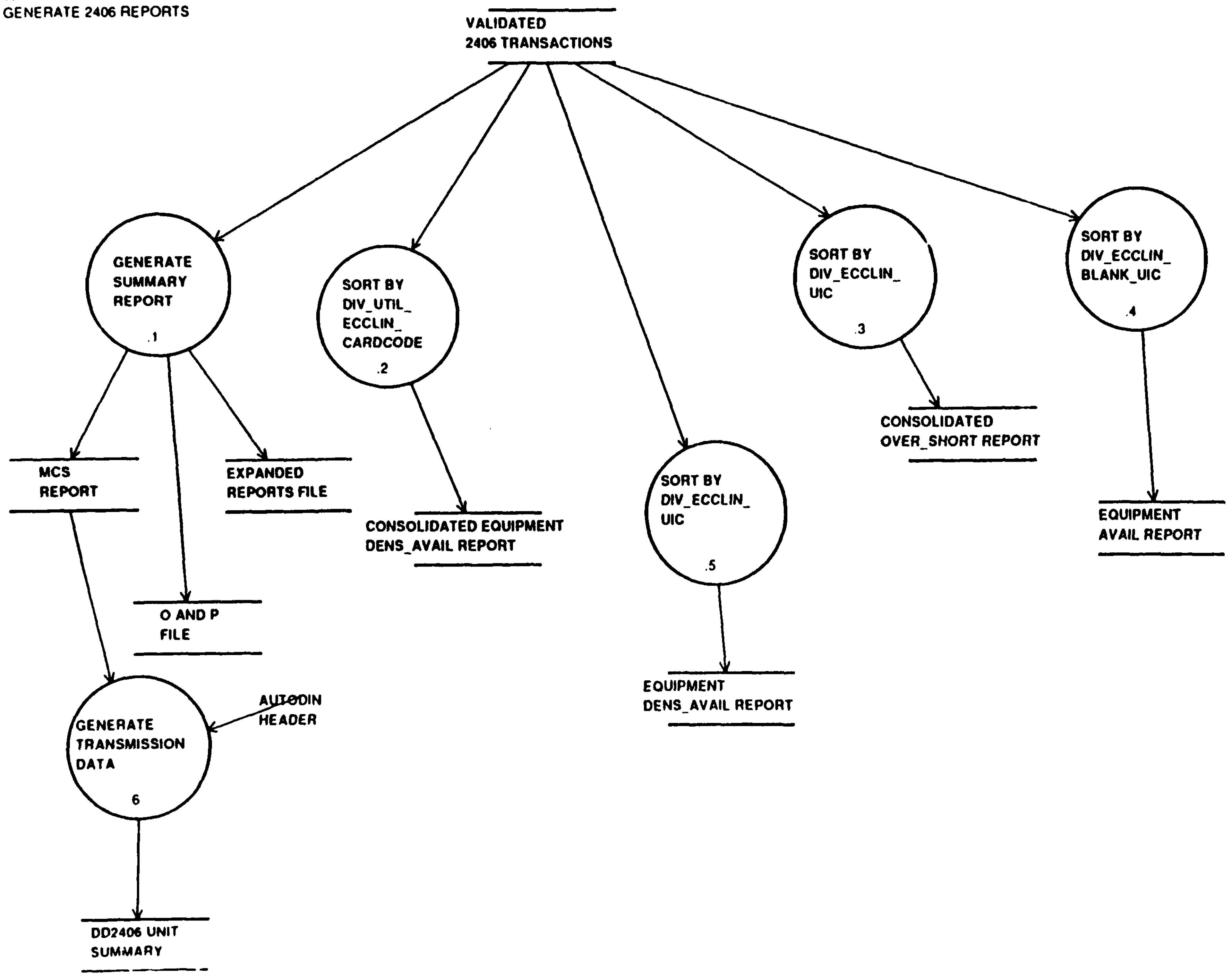
2:1
UPDATE ECC_LIN MASTER



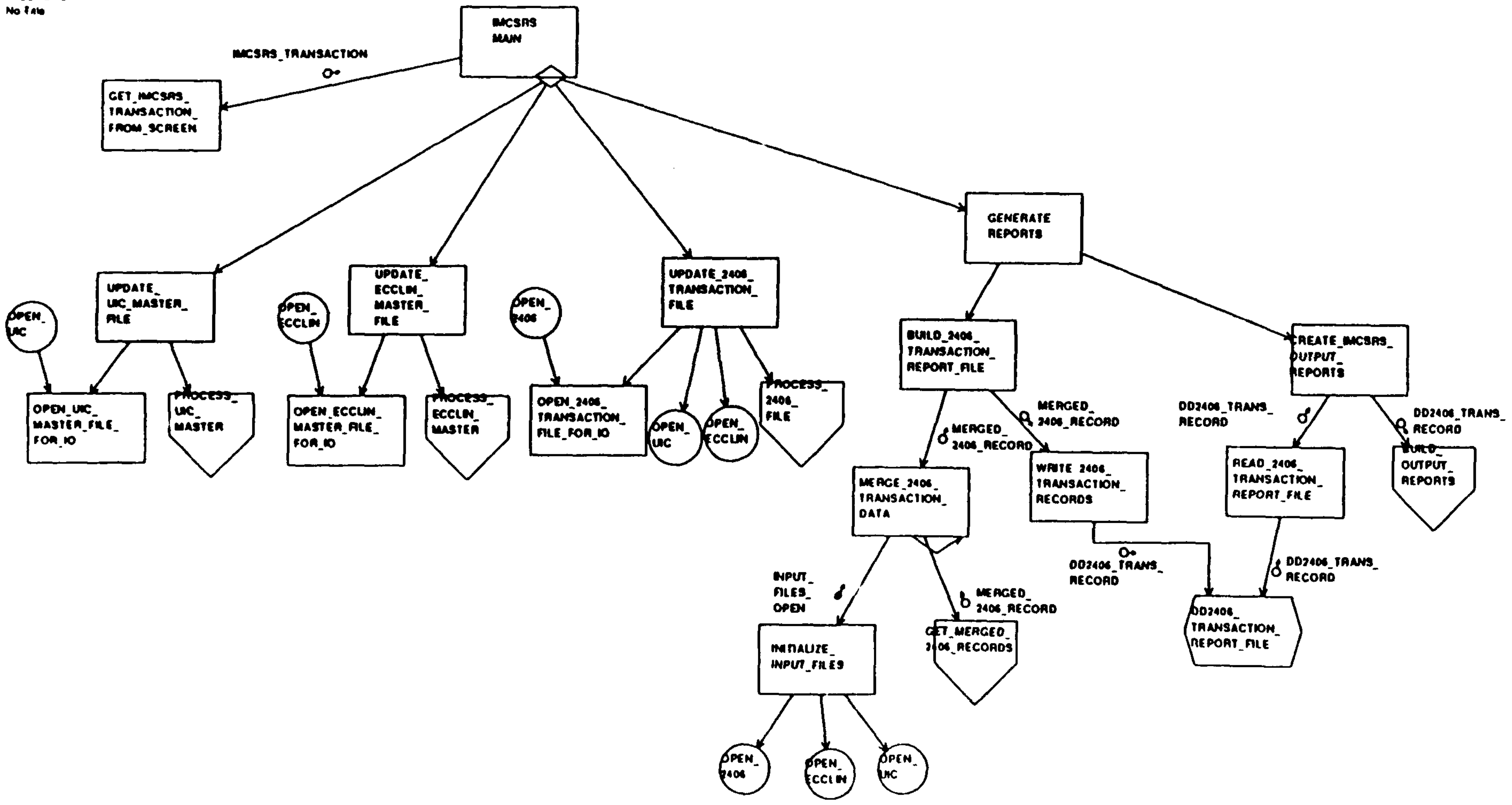
3.5
EDIT 2406 TRANSACTION FILE

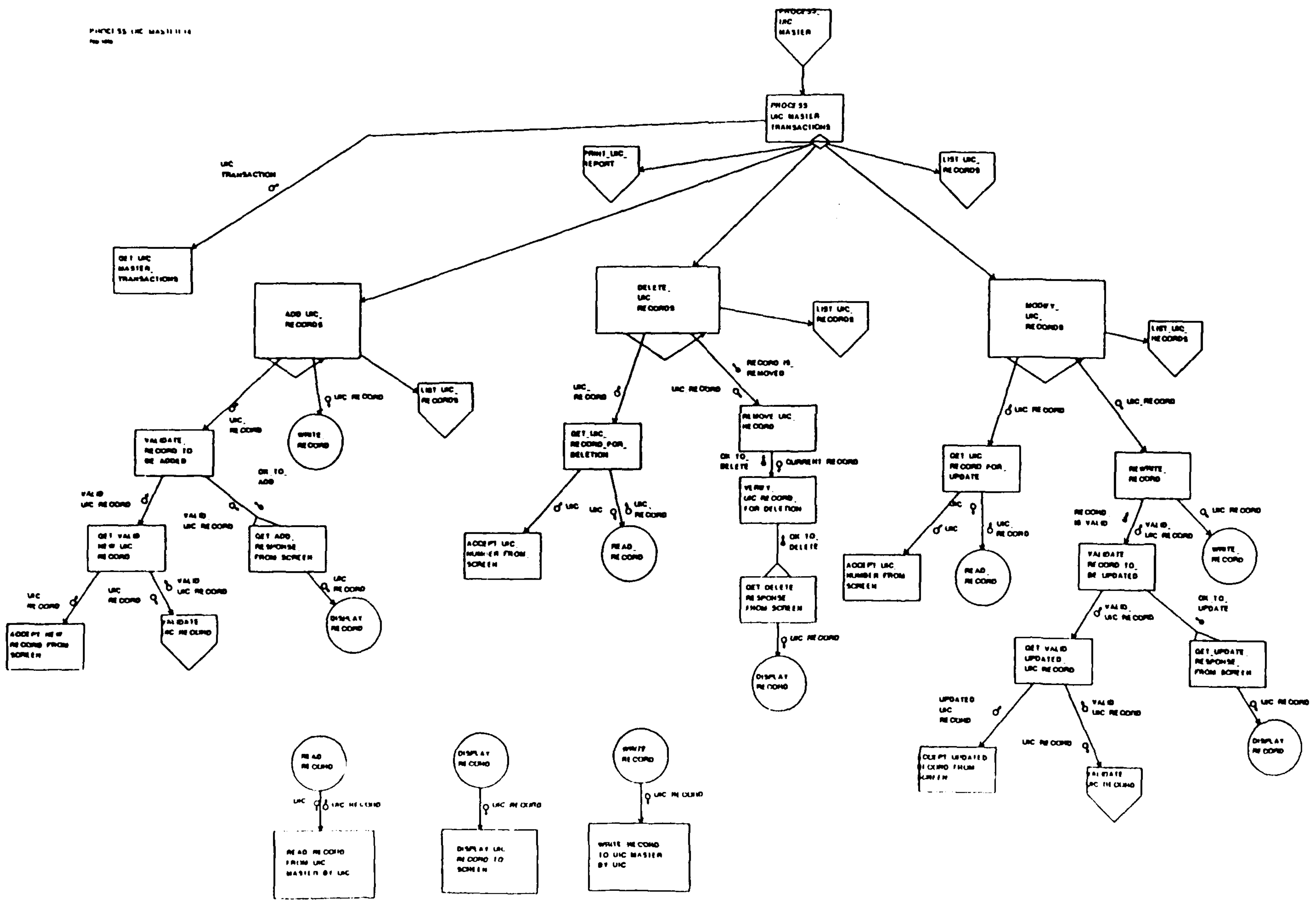

DD2406TRANSACTIONS
UIC MASTER
ECCLIN MASTER
VALIDATE DD2406 TRANSACTIONS .1
VALID 2406 TRANSACTIONS
EXPAND DD2406 TRANSACTIONS .2
DD2406 TRANSACTION FILE
DELETE 2406 TRANSACTIONS 3
ADD 2406 TRANSACTIONS 4
MODIFY 2406 TRANSACTIONS 5
DELETED 2406 TRANSACTIONS

5.4
GENERATE 2406 REPORTS

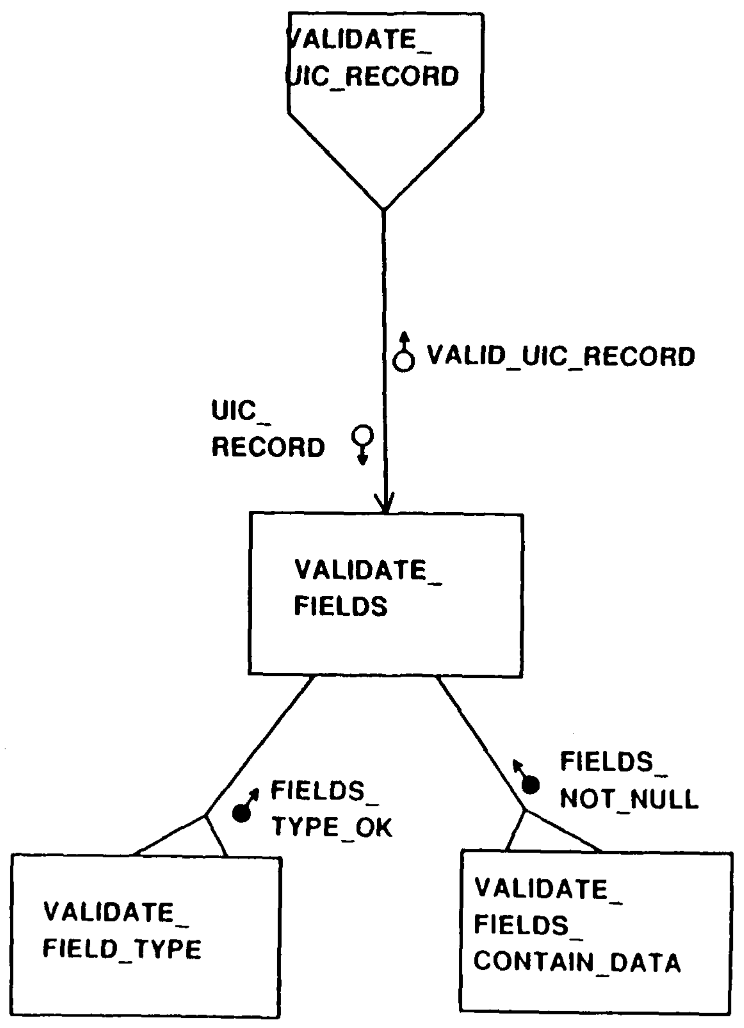

VALIDATED 2406 TRANSACTIONS
GENERATE SUMMARY REPORT .1
SORT BY DIV_UTIL_ ECCLIN_ CARDCODE .2
SORT BY DIV_ECCLIN_ UIC .3
SORT BY DIV_ECCLIN_ BLANK_UIC .4
SORT BY DIV_ECCLIN_ UIC .5
MCS REPORT
EXPANDED REPORTS FILE
O AND P FILE
CONSOLIDATED EQUIPMENT DENS_AVAIL REPORT
CONSOLIDATED OVER_SHORT REPORT
EQUIPMENT AVAIL REPORT
EQUIPMENT DENS_AVAIL REPORT
AUTODIN HEADER
GENERATE TRANSMISSION DATA 6
DD2406 UNIT SUMMARY

IMCSRS 12
No Title

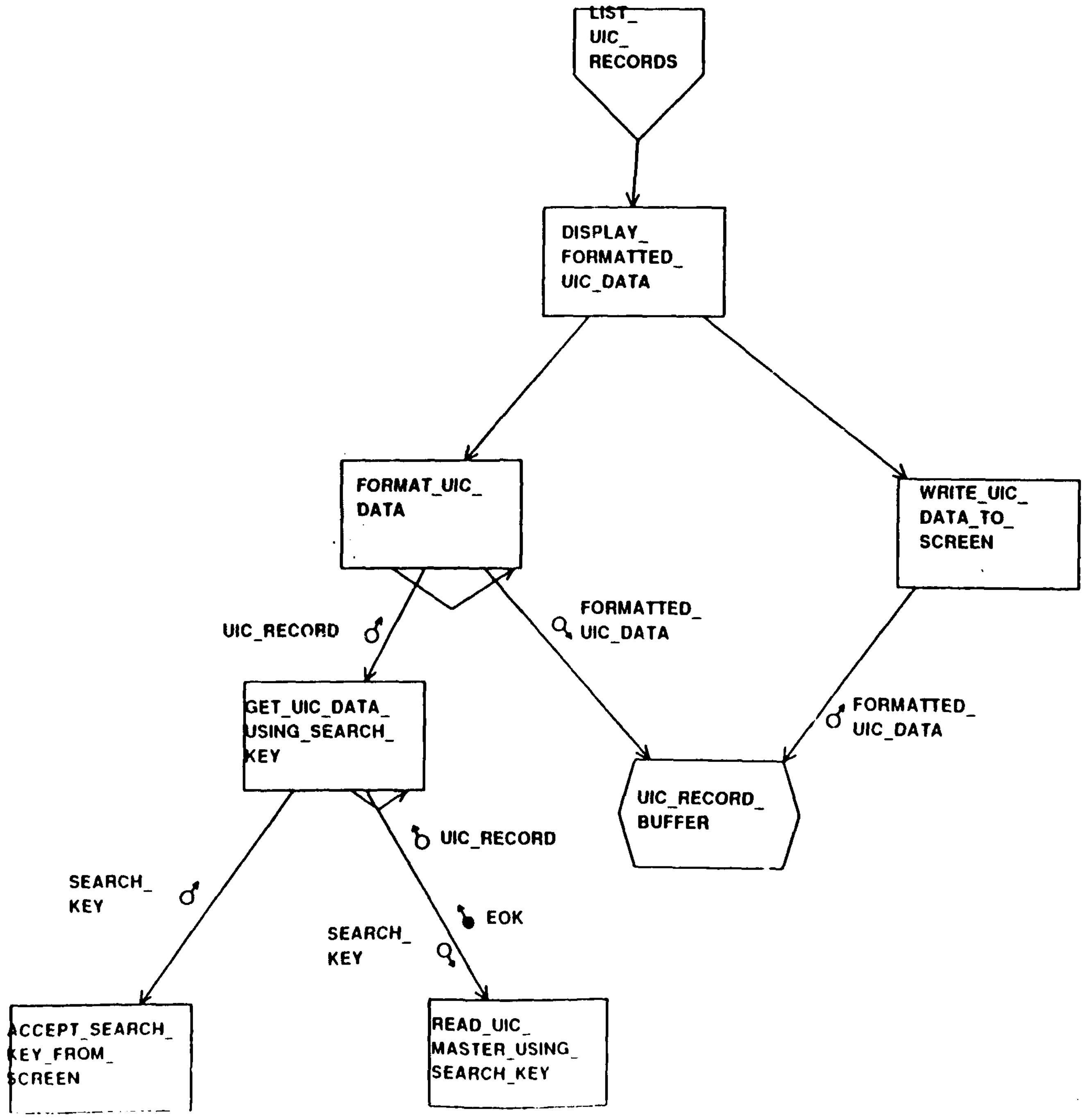



PROCESS UIC MASTER [illegible]
PROCESS_ UIC MASTER
PROCESS UIC MASTER TRANSACTIONS
UIC TRANSACTION
GET UIC MASTER_ TRANSACTIONS
PRINT_UIC_ REPORT
LIST UIC_ RECORDS
ADD UIC_ RECORDS
DELETE_ UIC RECORDS
MODIFY_ UIC_ RECORDS
LIST UIC_ RECORDS
LIST UIC_ RECORDS
LIST UIC_ RECORDS
UIC_ RECORD
UIC RECORD
WRITE RECORD
VALIDATE_ RECORD TO BE ADDED
OK TO_ ADD
VALID UIC RECORD
VALID UIC RECORD
GET VALID NEW UIC RECORD
GET ADD_ RESPONSE FROM SCREEN
UIC RECORD
UIC RECORD
VALID UIC RECORD
UIC RECORD
ACCEPT NEW RECORD FROM SCREEN
VALIDATE UIC RECORD
DISPLAY RECORD
UIC_ RECORD
GET_UIC_ RECORD_FOR_ DELETION
UIC
UIC
UIC_ RECORD
ACCEPT UIC_ NUMBER FROM SCREEN
READ_ RECORD
UIC RECORD
RECORD IS_ REMOVED
REMOVE UIC_ RECORD
OK TO_ DELETE
CURRENT RECORD
VERIFY_ UIC RECORD_ FOR DELETION
OK TO_ DELETE
GET DELETE RESPONSE FROM SCREEN
UIC RECORD
DISPLAY RECORD
UIC RECORD
UIC_RECORD
GET UIC_ RECORD FOR_ UPDATE
UIC
UIC
UIC_ RECORD
ACCEPT UIC_ NUMBER FROM SCREEN
READ_ RECORD
REWRITE_ RECORD
RECORD IS VALID
VALID_ UIC RECORD
UIC RECORD
WRITE_ RECORD
VALIDATE_ RECORD TO_ BE UPDATED
VALID_ UIC RECORD
OK TO_ UPDATE
GET VALID UPDATED_ UIC RECORD
GET_UPDATE_ RESPONSE_ FROM SCREEN
UIC RECORD
UPDATED UIC RECORD
VALID UIC RECORD
UIC RECORD
ACCEPT UPDATED RECORD FROM SCREEN
VALIDATE UIC RECORD
DISPLAY RECORD
READ RECORD
UIC
UIC RECORD
READ RECORD FROM UIC MASTER BY UIC
DISPLAY RECORD
UIC RECORD
DISPLAY UIC RECORD TO SCREEN
WRITE RECORD
UIC RECORD
WRITE RECORD TO UIC MASTER BY UIC

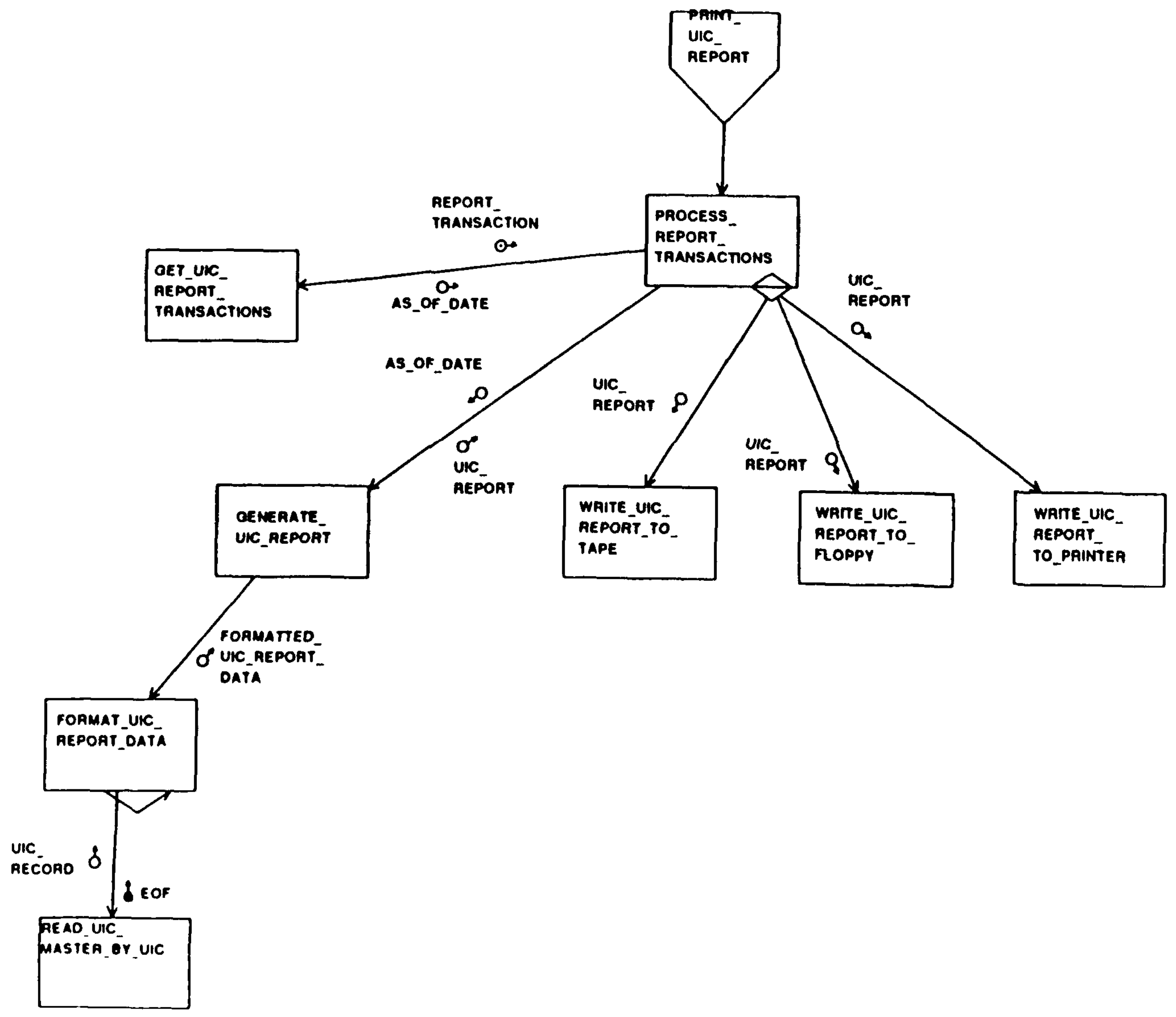
VALIDATE_UIC_RECORD;2
No title



LIST_UIC_RECORDS;2
No title

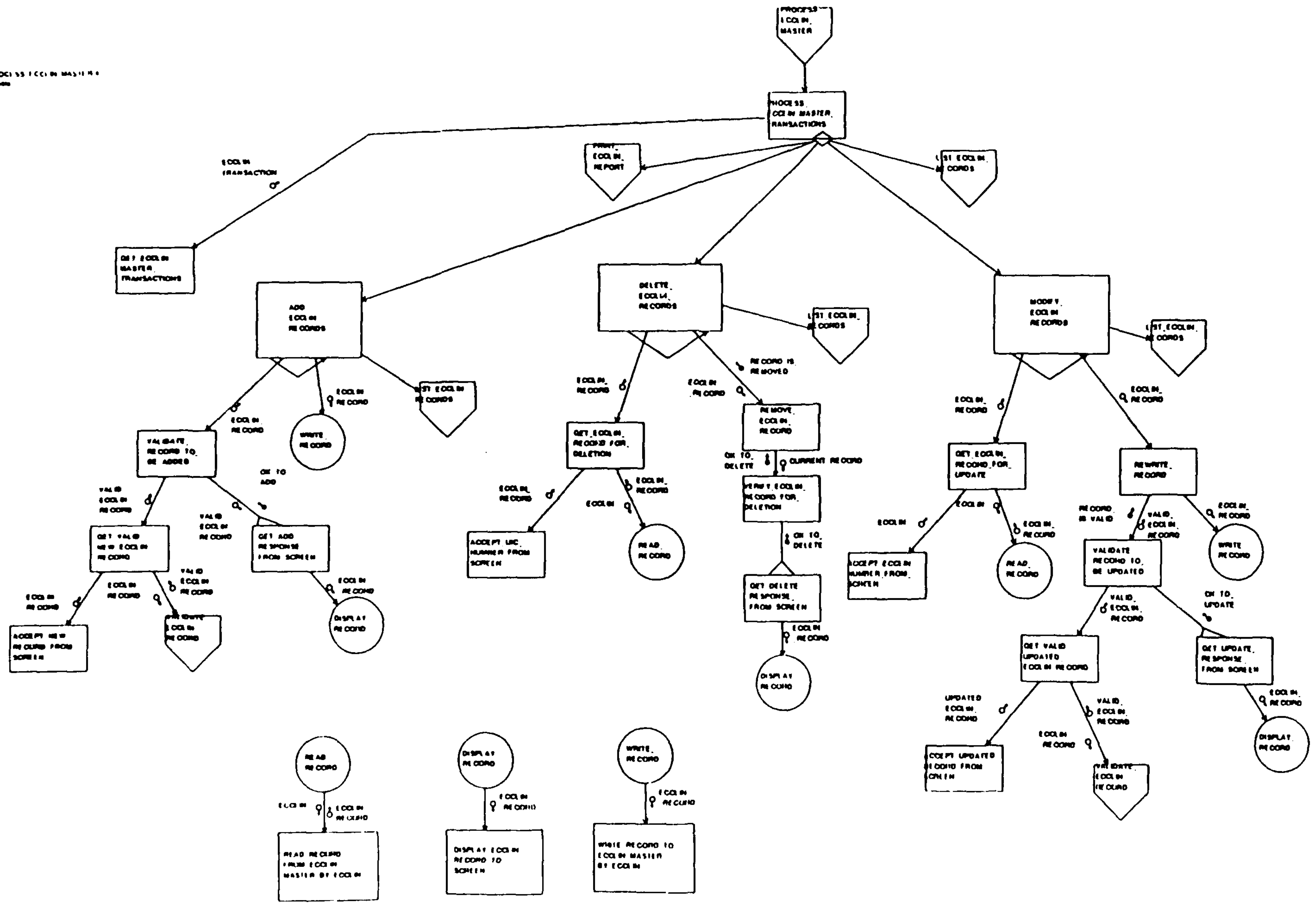

LIST_ UIC_ RECORDS
DISPLAY_ FORMATTED_ UIC_DATA
FORMAT_UIC_ DATA
WRITE_UIC_ DATA_TO_ SCREEN
UIC_RECORD
FORMATTED_ UIC_DATA
FORMATTED_ UIC_DATA
GET_UIC_DATA_ USING_SEARCH_ KEY
UIC_RECORD_ BUFFER
UIC_RECORD
SEARCH_ KEY
EOK
SEARCH_ KEY
ACCEPT_SEARCH_ KEY_FROM_ SCREEN
READ_UIC_ MASTER_USING_ SEARCH_KEY

PRINT UIC REPORT.1
No title



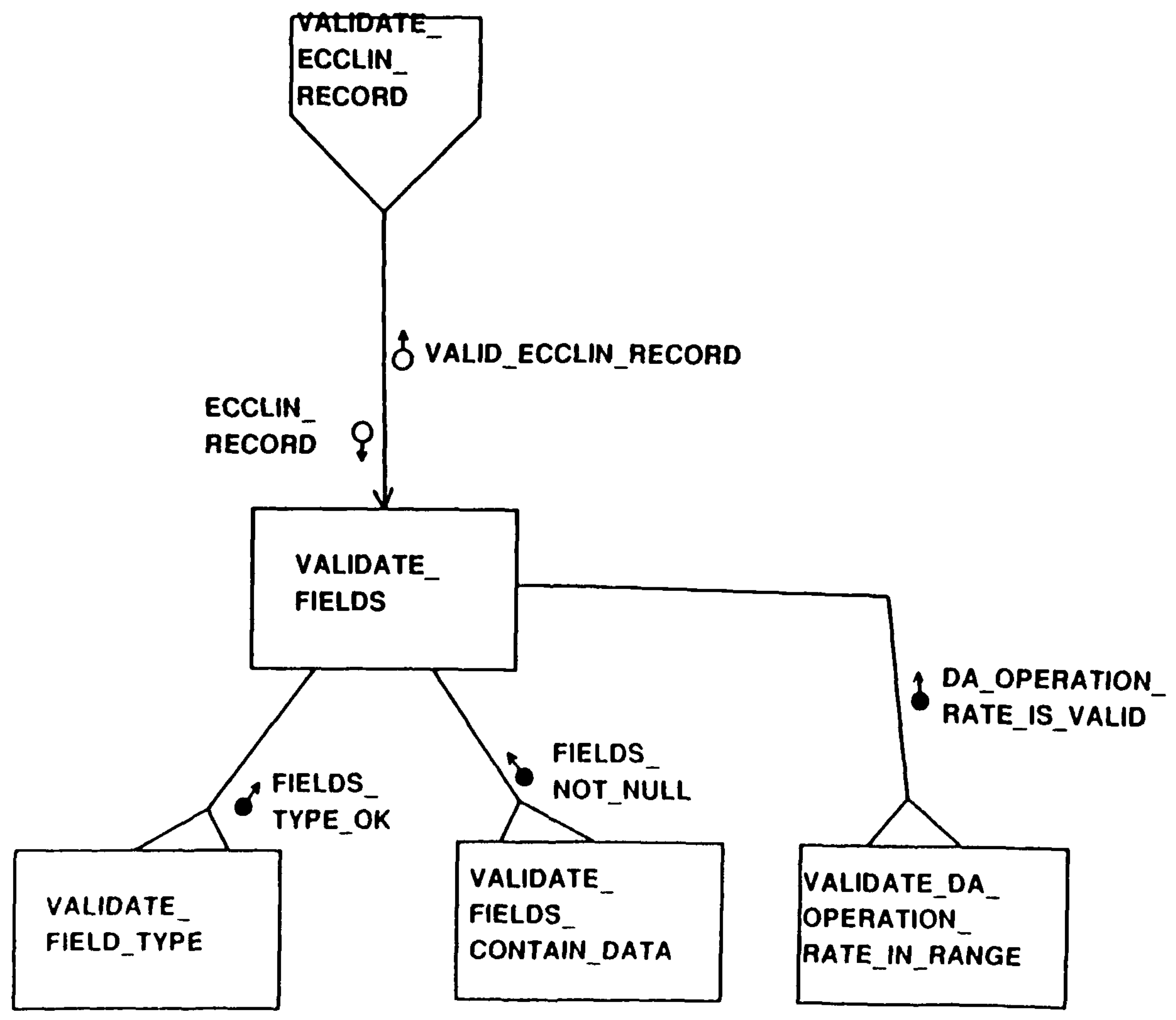
PROCESS ECCLN MASTER [illegible]

[illegible]

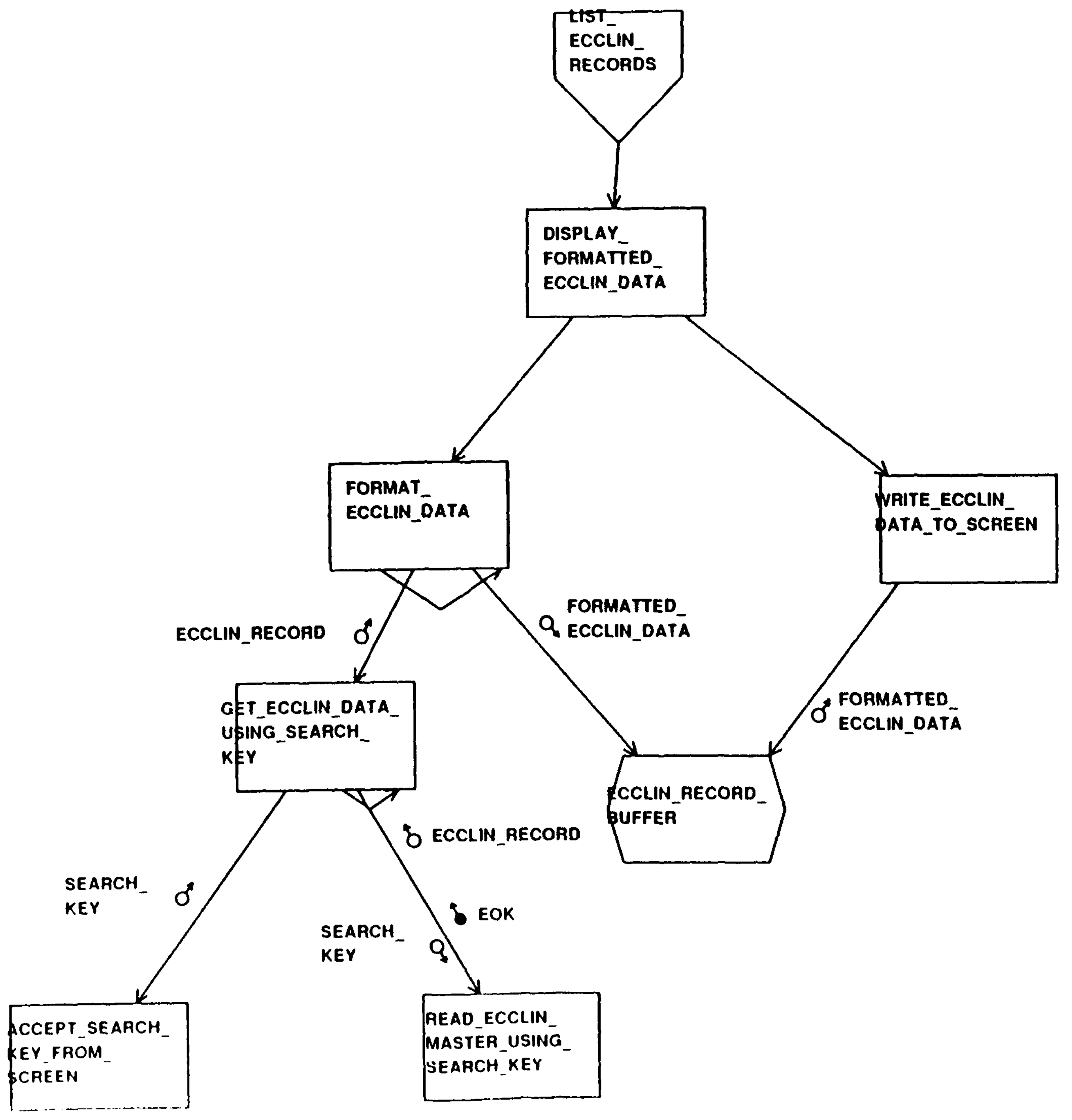


PROCESS ECCLN MASTER
PROCESS ECCLN MASTER TRANSACTIONS
ECCLN TRANSACTION
GET ECCLN MASTER TRANSACTIONS
PRINT ECCLN REPORT
LIST ECCLN RECORDS
ADD ECCLN RECORDS
DELETE ECCLN RECORDS
MODIFY ECCLN RECORDS
ECCLN RECORD
WRITE RECORD
VALIDATE RECORD TO BE ADDED
VALID ECCLN RECORD
OK TO ADD
GET VALID NEW ECCLN RECORD
GET ADD RESPONSE FROM SCREEN
ACCEPT NEW RECORD FROM SCREEN
VALIDATE ECCLN RECORD
DISPLAY RECORD
RECORD IS REMOVED
GET ECCLN RECORD FOR DELETION
REMOVE ECCLN RECORD
OK TO DELETE
CURRENT RECORD
ECCLN
ACCEPT UIC NUMBER FROM SCREEN
READ RECORD
VERIFY ECCLN RECORD FOR DELETION
GET DELETE RESPONSE FROM SCREEN
GET ECCLN RECORD FOR UPDATE
REWRITE RECORD
RECORD IS VALID
ACCEPT ECCLN NUMBER FROM SCREEN
VALIDATE RECORD TO BE UPDATED
OK TO UPDATE
GET VALID UPDATED ECCLN RECORD
GET UPDATE RESPONSE FROM SCREEN
UPDATED ECCLN RECORD
ACCEPT UPDATED RECORD FROM SCREEN
READ RECORD FROM ECCLN MASTER BY ECCLN
DISPLAY ECCLN RECORD TO SCREEN
WRITE RECORD TO ECCLN MASTER BY ECCLN

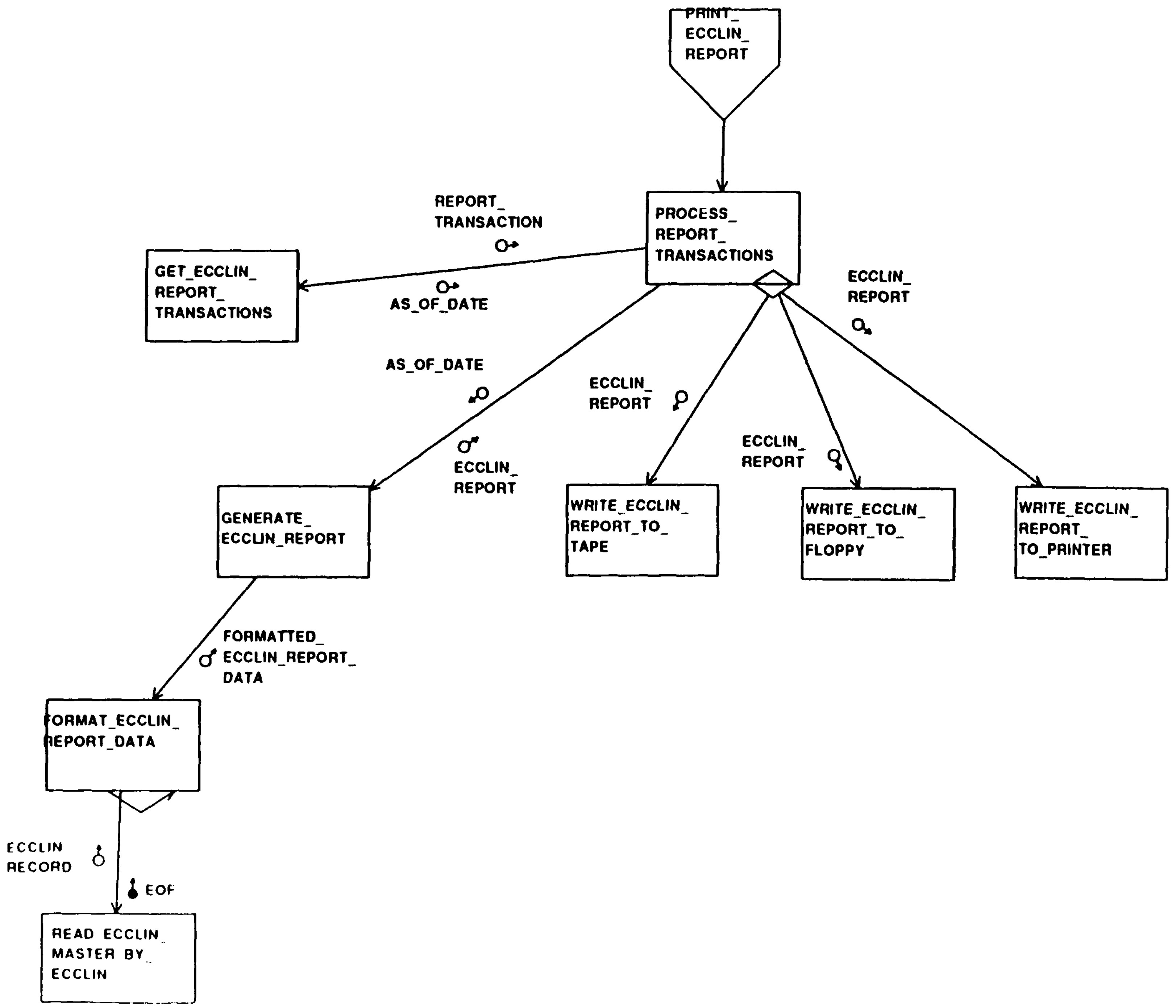
VALIDATE_ECCLIN_RECORD;2
No title



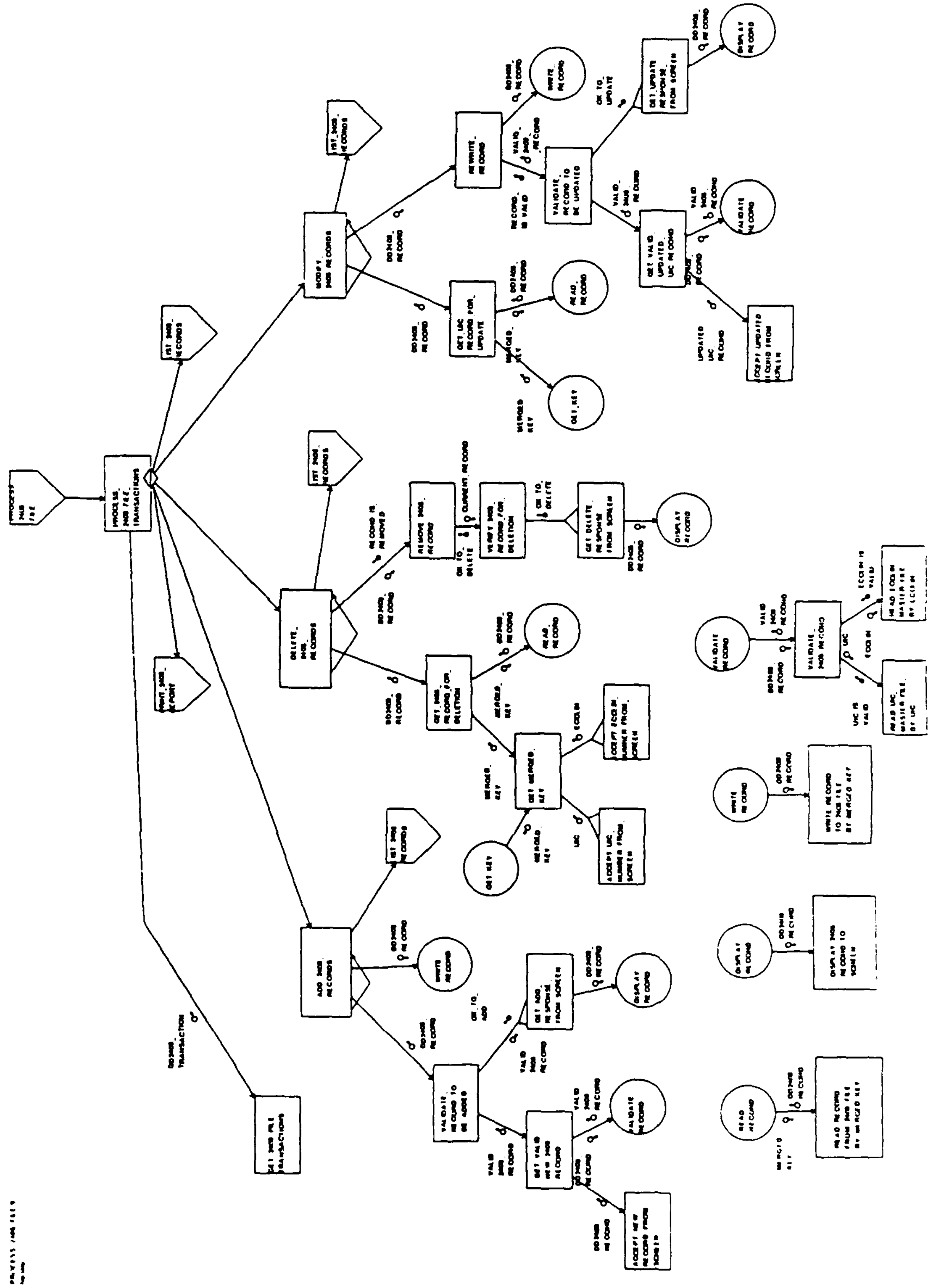
LIST_ECCLIN_RECORDS.2
No title


LIST_ ECCLIN_ RECORDS
DISPLAY_ FORMATTED_ ECCLIN_DATA
FORMAT_ ECCLIN_DATA
WRITE_ECCLIN_ DATA_TO_SCREEN
ECCLIN_RECORD
FORMATTED_ ECCLIN_DATA
FORMATTED_ ECCLIN_DATA
GET_ECCLIN_DATA_ USING_SEARCH_ KEY
ECCLIN_RECORD_ BUFFER
ECCLIN_RECORD
SEARCH_ KEY
EOK
SEARCH_ KEY
ACCEPT_SEARCH_ KEY_FROM_ SCREEN
READ_ECCLIN_ MASTER_USING_ SEARCH_KEY

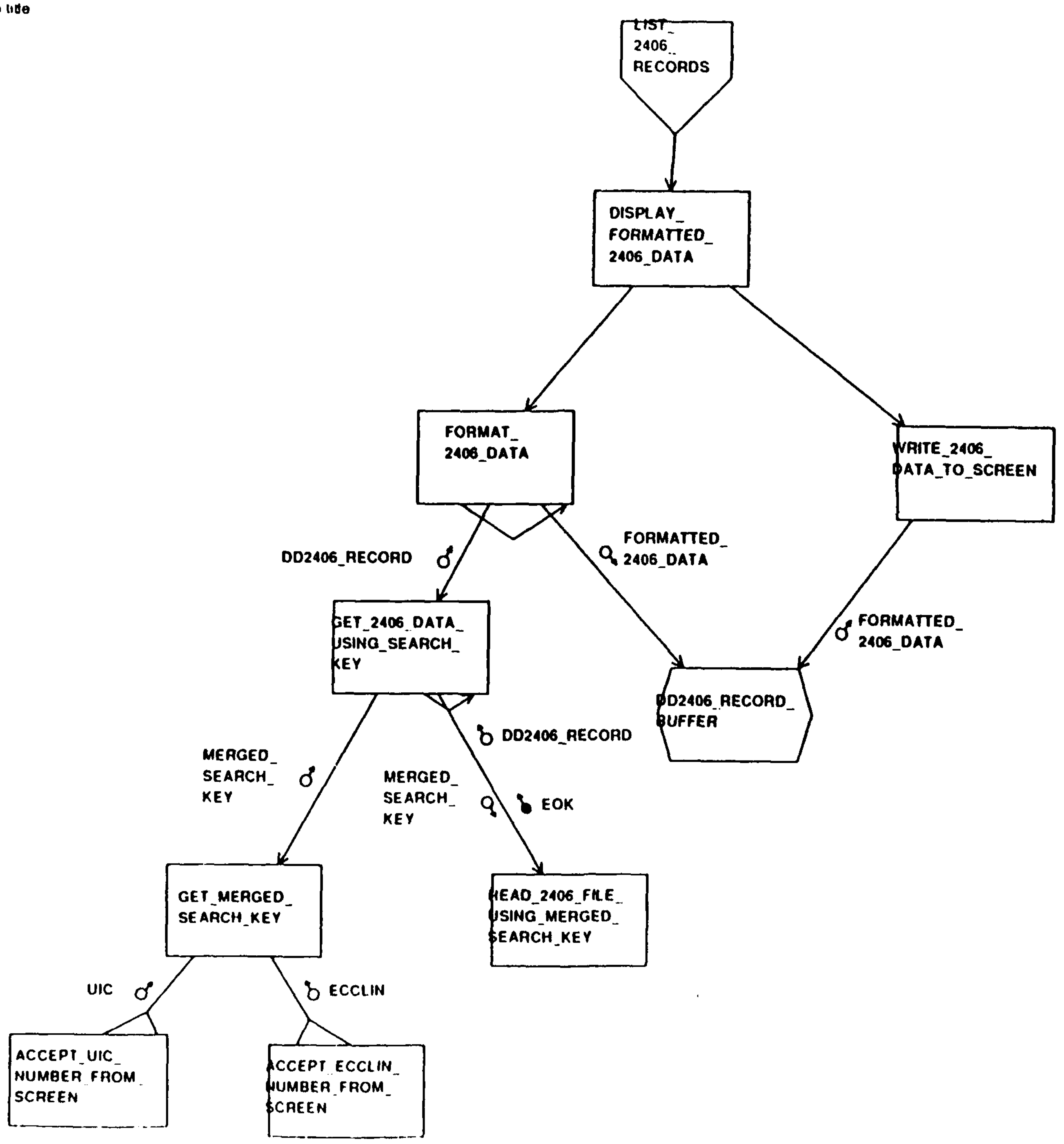
PRINT_ECCLIN_REPORT;2
No title



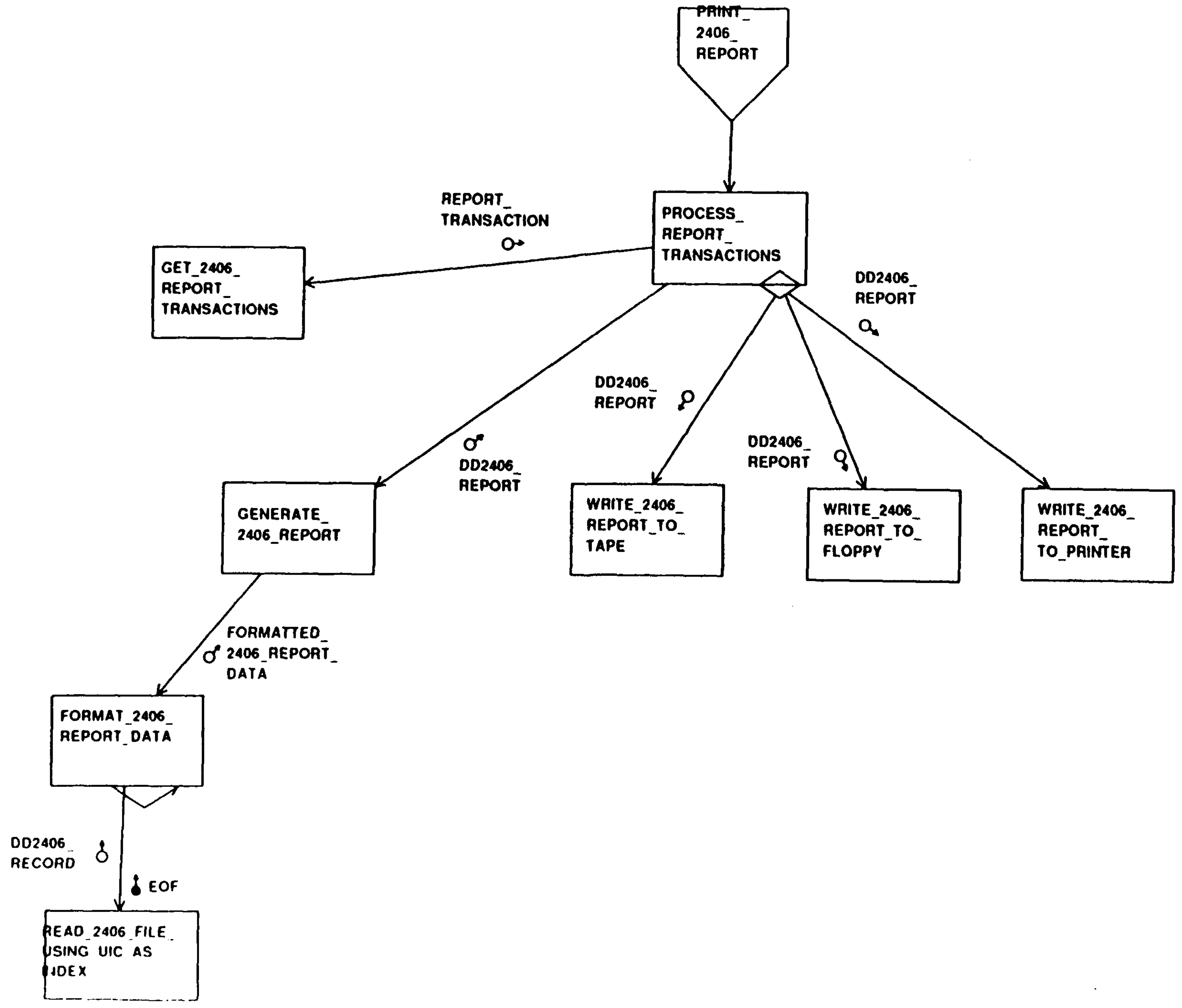
COPY AVAILABLE TO DTIC DOES NOT PERMIT FULLY LEGIBLE REPRODUCTION



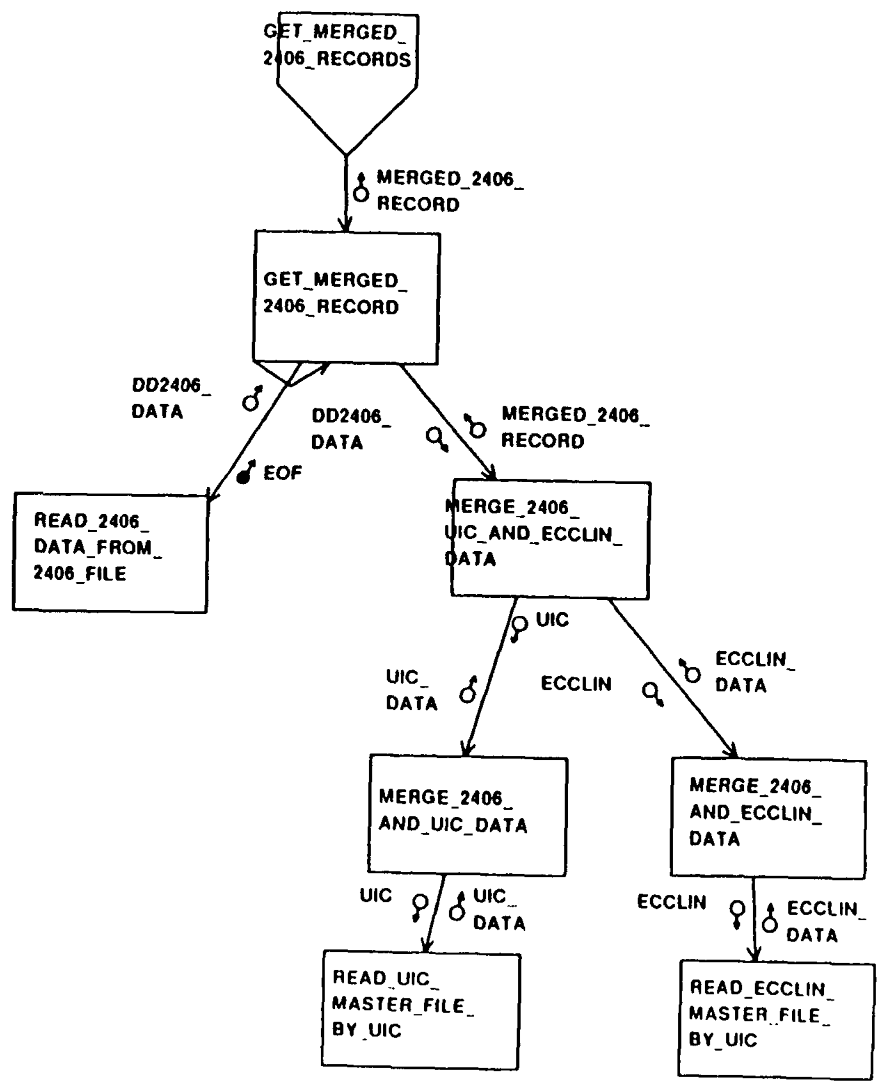
LIST_2406_RECORDS.1
No title



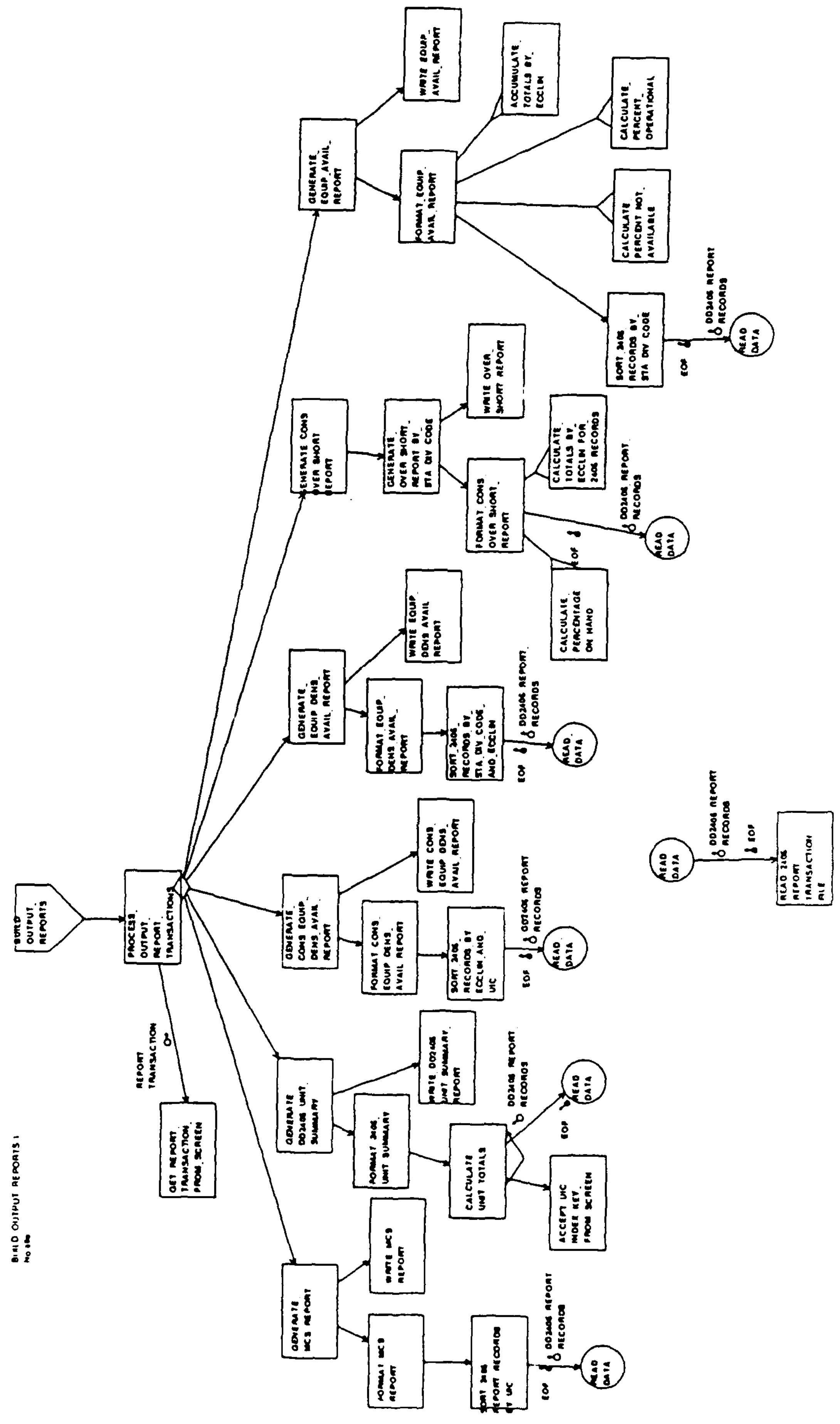
PRINT_2406_REPORT;1
No title



GET_MERGED_2406_RECORDS.4
No title



COPY AVAILABLE TO DTIC DOES NOT PERMIT FULLY LEGIBLE REPRODUCTION


BUILD OUTPUT REPORTS
PROCESS OUTPUT REPORT TRANSACTIONS
REPORT TRANSACTION
GET REPORT TRANSACTION FROM SCREEN
GENERATE EQUIP AVAIL REPORT
WRITE EQUIP AVAIL REPORT
FORMAT EQUIP AVAIL REPORT
ACCUMULATE TOTALS BY ECCLIN
CALCULATE PERCENT OPERATIONAL
CALCULATE PERCENT NOT AVAILABLE
SORT 2406 RECORDS BY STA DIV CODE
DD2406 REPORT RECORDS
EOF
READ DATA
GENERATE CONS OVER SHORT REPORT
GENERATE OVER SHORT REPORT BY STA DIV CODE
WRITE OVER SHORT REPORT
FORMAT CONS OVER SHORT REPORT
CALCULATE TOTALS BY ECCLIN FOR 2406 RECORDS
CALCULATE PERCENTAGE ON HAND
GENERATE EQUIP DEMS AVAIL REPORT
WRITE EQUIP DEMS AVAIL REPORT
FORMAT EQUIP DEMS AVAIL REPORT
SORT 2406 RECORDS BY STA DIV CODE AND ECCLIN
READ 2406 REPORT TRANSACTION FILE
GENERATE CONS EQUIP DEMS AVAIL REPORT
WRITE CONS EQUIP DEMS AVAIL REPORT
FORMAT CONS EQUIP DEMS AVAIL REPORT
SORT 2406 RECORDS BY ECCLIN AND UIC
GENERATE DD2406 UNIT SUMMARY
WRITE DD2406 UNIT SUMMARY REPORT
FORMAT 2406 UNIT SUMMARY
CALCULATE UNIT TOTALS
ACCEPT UIC INDEX KEY FROM SCREEN
GENERATE MCS REPORT
WRITE MCS REPORT
FORMAT MCS REPORT
SORT 2406 REPORT RECORDS BY UIC
BUILD OUTPUT REPORTS 1